17th August, 2016

classmate
Date
Page 1

* Compiler :-

- Introduction :-

A compiler is a program that can read a program in one language i.e. source language and translate it into equivalent program in another language i.e. target language.

Source Program.

Compiler Error Message.

Target Program.

Fig: Compiler.

Qs) What is the difference between compiler & interpreter?

H/W Ans:

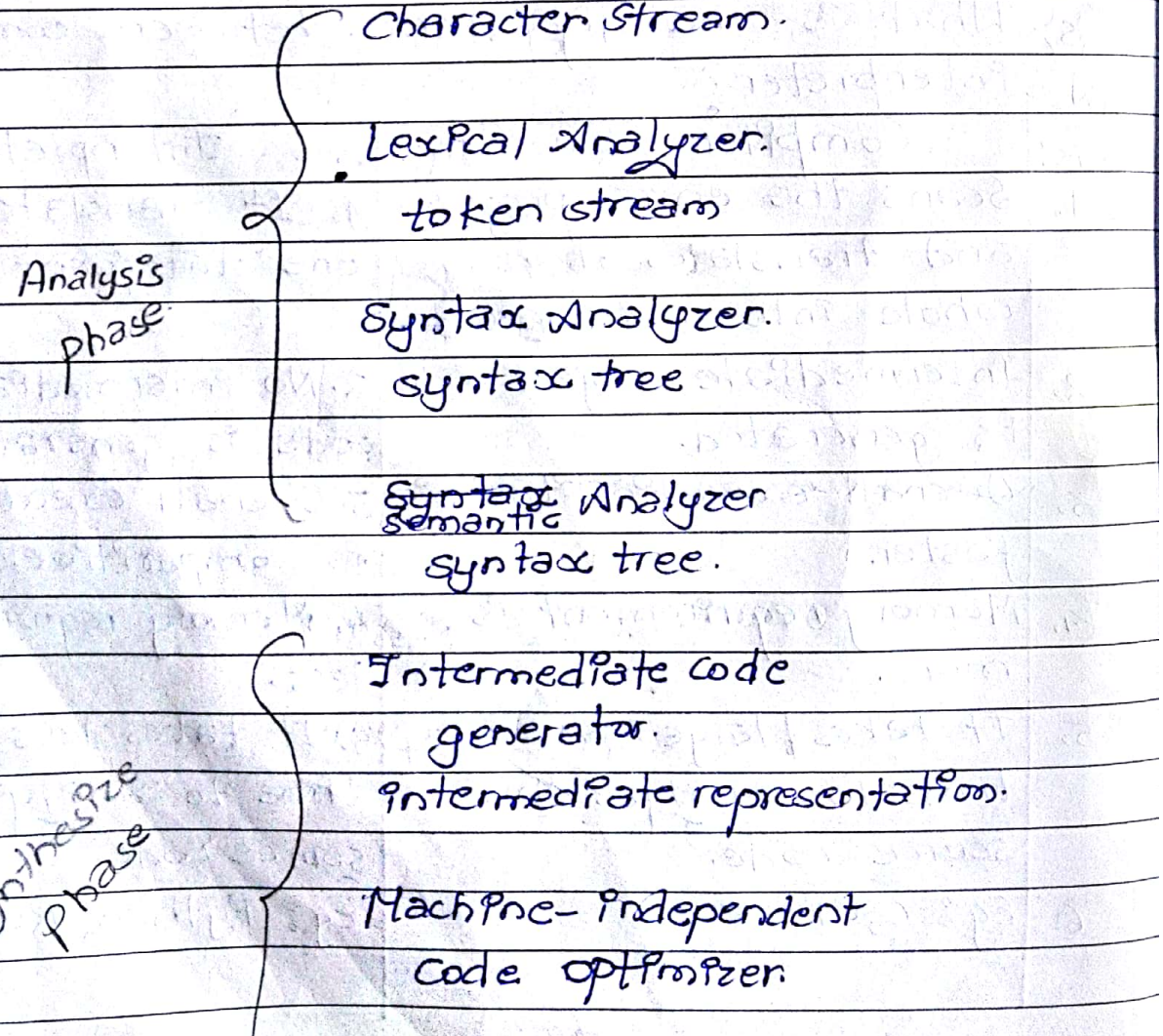
	Compiler	Interpreter
1.	Scans the entire program and translates it as a whole into machine code.	1. It translates a program one statement at a time.
2.	Intermediate object code is generated.	2. No intermediate object code is generated.
3.	Overall execution time is faster.	3. Overall execution time is comparitively slower.
4.	Memory requirement is more.	4. Memory requirement is less.
5.	It takes large amount of time to analyze the source code.	5. It takes less amount of time to analyze the source code.
6.	Eg: C, C++.	6. Eg: Python, Ruby, Basic.

* Computer / Program System :-

program.
↓
Preprocessor
↓
compiler.
↓
Assembler
↓
Loader / Linker.

Q5) Define compiler. Describe the phases of the compiler.
VVoJmp
T.U. 2068

* Phases of compiler.



Intermediate representation.
code generator
target machine code
Machine dependent
code optimizer
target machine code.

Fig:- Phases of compiler.

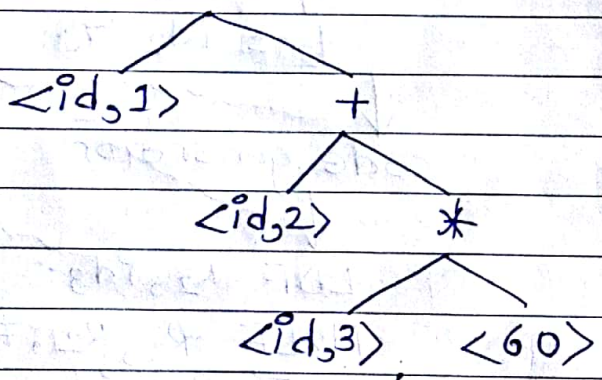
Example:-

Position = initial + rate * 60 (character stream)

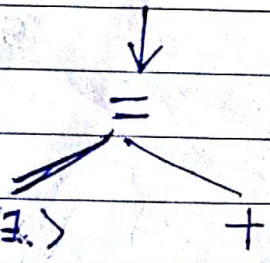
↓ Lexical Analyzer

token ← { <id,1> <=> <id,2> <+> <id,3> <*> <60> }

↓ Syntax Analyzer



↓ (type conversion) Semantic Analyzer



Intermediate representation.
code generator
target machine code
Machine dependent
code optimizer
target machine code.

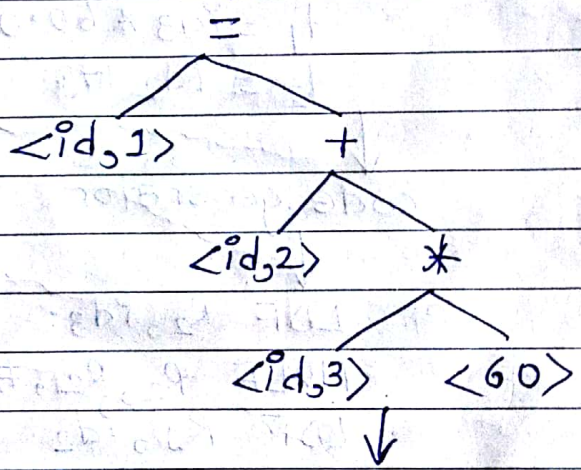
Fig:- Phases of compiler.

Example:-
Position = initial + rate * 60 (character stream)

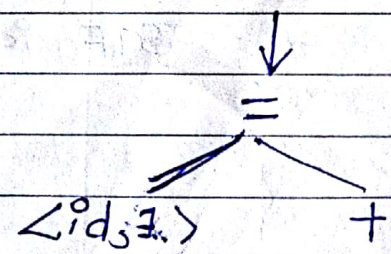
↓ Lexical Analyzer

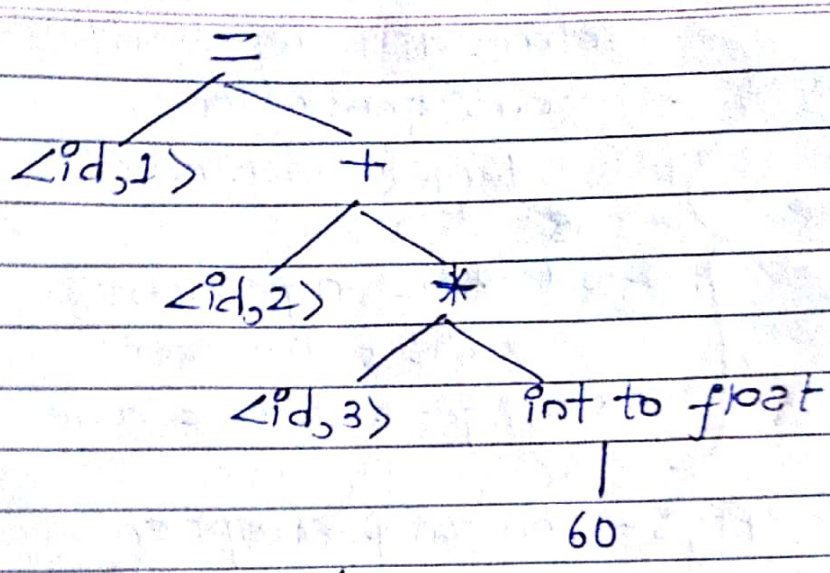
token ← { <id,1> <=> <id,2> <+> <id,3> <*> <60>

↓ Syntax Analyzer



↓ (type conversion) Semantic Analyzer





↓

Intermediate code generator

↓

t_1 = int to float (60)
t_2 = id3 * t_1.
t_3 = id2 + t_2
id1 = t_3.

↓

code optimizer.

↓

t_1 = id3 * 60.0
t_2 = id2 + t_1.

↓

code generator.

↓

```
LDF  R2, id3.
MVLF R2, R2, # 60.0.
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1.
```

21st August

Example:

* Input stream:

E = ((a+b) * (c-d) / (e+f))

Pass into phases of compiler.

Character stream.

↓

((a+b) * (c-d) / (e+f)).

↓

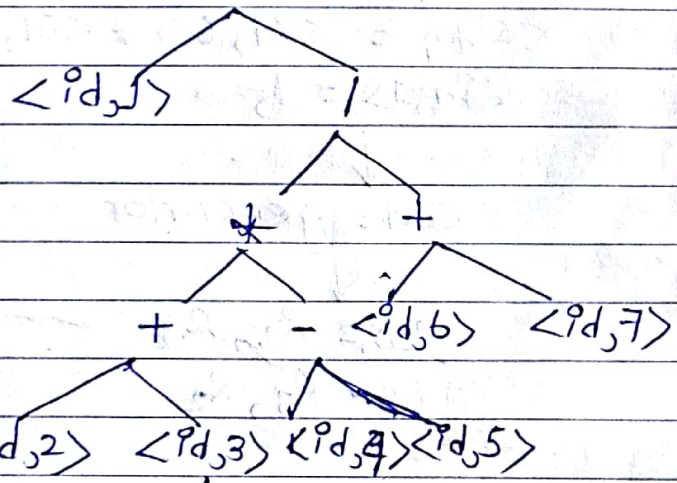
Lexical Analyzer

<id,1> <=> <id,2> <+> <id,3> * <id,4> <-> <id,5>
</> <id,6> <+> <id,7>.

↓

Syntax Analyzer.

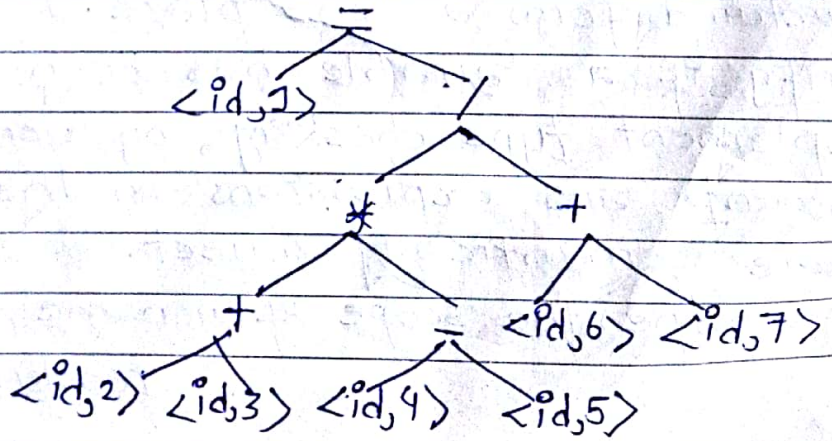
↓



↓

Semantic Analyzer.

↓



↓

Intermediate code generator

↓

$t_1 = \langle id, 4\rangle - \langle id, 5\rangle$
$t_2 = \langle id, 2\rangle + \langle id, 3\rangle$
$t_3 = t_1 * t_2$
$t_4 = \langle id, 6\rangle + \langle id, 7\rangle$
$t_5 = t_3 / t_4$
$\langle id, 1\rangle = t_5$

↓

code optimizer

↓

$t_1 = \langle id, 4\rangle - \langle id, 5\rangle$
$t_2 = \langle id, 2\rangle + \langle id, 3\rangle$
$t_3 = t_1 * t_2$
$t_4 = \langle id, 6\rangle + \langle id, 7\rangle$
$\langle id, 1\rangle = t_5$

↓

code generator

↓

LDF R_2, R_1
LDF R_3, R_2

V.V.Imp

T.U. 2068

* Role of Symbol table :-

(i) To store the names of all the entities in a structured form at one place.

(ii) To verify if a variable has been declared.

(iii) To implement type checking, by verifying assignments and expressions in the source code are semantically correct.

(iv) To determine the scope of a name (scope resolution)

22nd August

→ can be either linear or hash table.
→ implemented as hash table.

* Symbol Table:- → source code are itself treated as key for hash function.
It is an essential data structure used by the compiler to remember information about identifiers appearing in the source language program.
- It contains nearly all the information needed by the different phases of compiler.
- The types of symbols that are stored in the symbol table include variables, procedures, defined constants, labels structures, etc.
- Maintains an entry for each name in following format.

Name	type	location
x	integer	offset of x
y	"	" " y
z	float	" " z
⋮	⋮	⋮

Eg: static int interest.
→ <interest, int, static>

Fig: Symbol Table.

T.U Qs) Define different phases of compiler with practical example. Write down the difference between lifetime and visibility of a variable.

* Information in Symbol Table:-
(i) Name (ii) Type (iii) Location (iv) Scope
(v) other attributes.

- Usage of symbol table and information:-
(i) Semantic analysis (ii) Code Generation.
(iii) Error detection. (iv) Optimization.

- Features of symbol table:-
(i) Lookup table (ii) Insert (lexical & syntax does)
(iii) Modify (iv) Delete.

Q5) Define different compiler construction tools.

23 3rd August

* Stack Machine :-
  - Contrast to an ordinary processor which use registers, a stack machine uses stack.
  - Stack does not need addressing as it is implicit in the operators which use stack.
  - Any expression can be transformed into postfix order and stack can be used to evaluate that expression without the need for explicitly locating any variables.

Example:

B+C-D (infix)

BC+D- (postfix)

Stack Machine :
```
push B
push C
add
push D
sub.
```

Using registers :-
```
load r0, B
load r1, C
add r0, r1 ; ; r0 + r1 → r0
load r2, D
sub r0, r2 ; ; r0 - r2 → r0.
```

* Syntax Definations:

Generally, context free grammar are used to specify the syntax of a language.

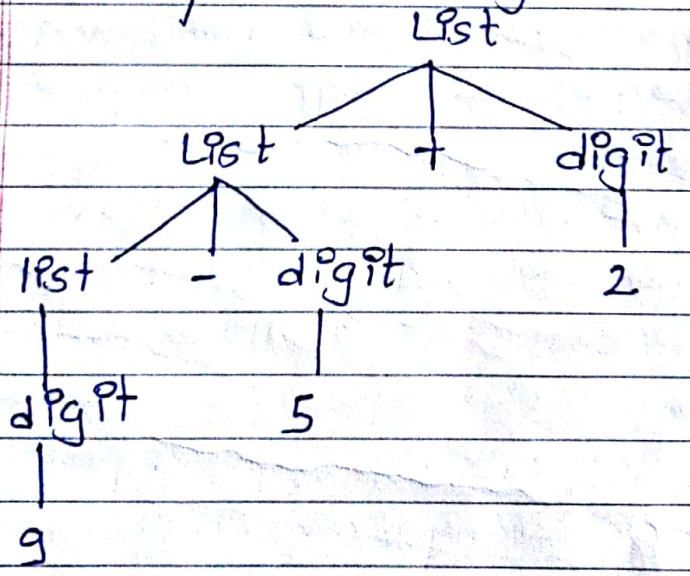
Example: if (expression) statement else statement
statement → ~~else~~ if (expression) stmt else stmt
where, if, else and parenthesis are terminals and expression and stmt are non-terminals.

Qs) Construct a grammar that describes the expression consisting of digits, plus and minus signs.

list → list + digit.
list → list - digit.
list → digit.
digit → 0|1|2|3|4|5|6|7|8|9

Ans: list → list + digit.
→ list + 2
→ list - digit + 2
→ list - 5 + 2
→ digit - 5 + 2
→ 9 - 5 + 2

The parse tree for the expression:



5> Code optimization :-
Code optimization phase attempts to improve the intermediate code, so that faster running machine code will result. It reduce the size of program.

6> Target code generator :-
- The final phase of compiler is the generation of target code, consisting normally of relocate machine code or assembly code.
- Memory locations are selected for each of the variables used by the program.
- Then, the each intermediate instruction is ~~related~~ translated into a sequence of machine instructions that perform the same task.

* Compiler Construction Tools :-
Compiler construction tools uses specalized language for specifying and implementing specific components and may many use quiet sophisticated algorithm. The most successful tools are those that hide the details of the generation algorithm and produce components that can be ~~easily~~ easily integrated into the remainder of the compiler.

- Some commonly used compiler construction tools are:

(1) Parser Generator :-
That automatically produce syntax analyzers from a grammatical description.

(VI) Compiler construction Toolkits :- that provides an integrated set of routines for constructing various phases of compiler.

of a programming language.

(II) Scanner Generator :-
That produce lexical analyzer from a regular expression decription of the tokens of a language

(III) Syntax-directed translation engines :-
That produce collection of routines for walking a parse tree and generating intermediate code.

(IV) code-generator generators :-
It produce a code generation from a collection of rules ~~from~~ for translating each operation of the intermediate language into the machine language for a target machine.

(V) Data flow analysis engines :-
That facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data flow analysis is a key part of code optimization.

* A simple one pass compiler :-
In computing programming, a one-pass compiler is a compiler that passes through the part of each compilation unit only once, immediately translating each part into its final machine code.

- Multipass compiler converts the program into one or more intermmediate representation between source code and machine code.

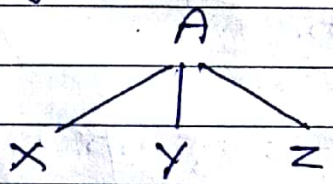
- Advantages:-
- One-pass compilers are smaller and faster than multi-pass compilers.

- Disadvantage:-
- One-pass compilers are unable to generate as efficient ~~as~~ program as multi-pass compilers due to limited scope of available information.

28th August

* Parse Tree:

A parse tree pictorically shows the start symbol of a grammar derives a ~~sting~~ string in a language. If non terminal 'A' has a production $A \rightarrow XYZ$ then a parse tree may have an interior node labelled 'A' with three children labelled X, Y, and Z from left to right.



A parse tree is a tree with the following properties:

(i) The root is labelled by the start symbol.
(ii) Each leaf is labelled by a token.
(iii) Each interior node is labelled by a non-termin

* Ambiguity :-

Ambiguity is the feature of any grammar in which there is more than one derivation for the same terminals strings using left most (right most derivation) i.e. there is two or more than two distinct parse tree for the same string.

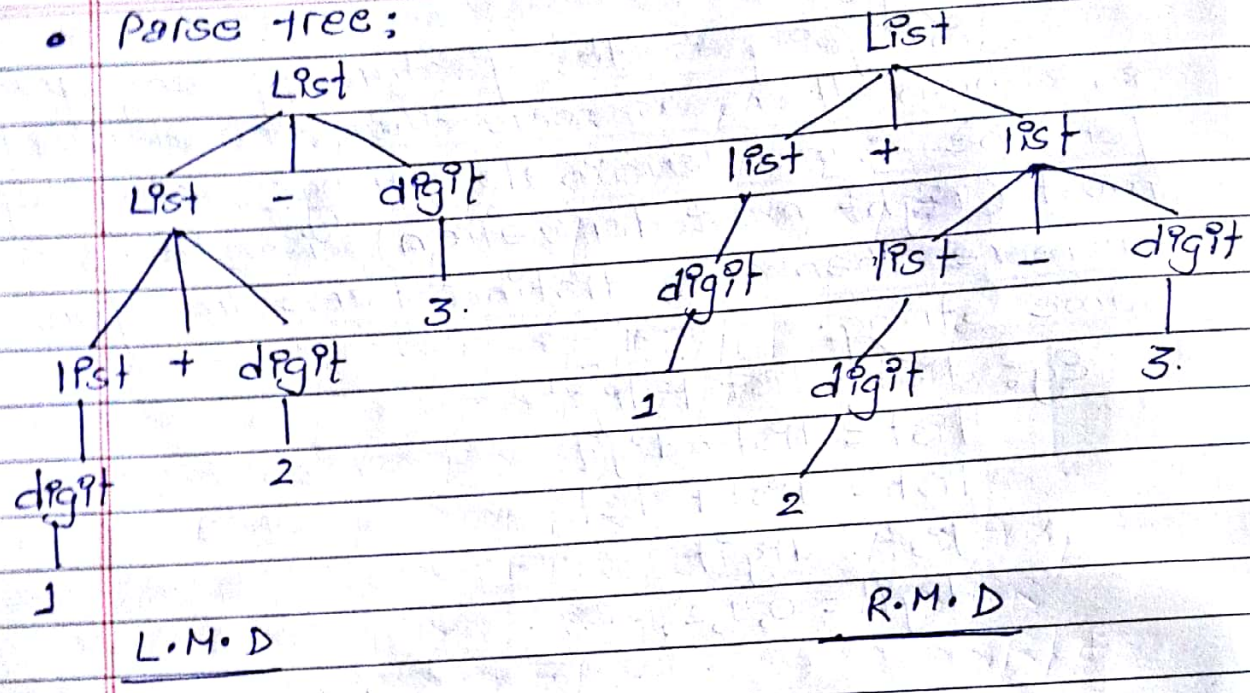
Eg: list = list + digit
list = list - digit
list = list * list
list = digit
digit = 0, 1, 2, ... 9

Expression : 1+2-3.

list → list - digit
(Leftmost → list + digit - digit
Derivative) → digit + digit - digit
→ 1 + digit - digit
→ 1 + 2 - 3

list → list + list
(Rightmost → list + list - digit
Derivative) → list + list - 3.
→ list + digit - 3
→ list + 2 - 3
→ digit + 2 - 3
→ 1 + 2 - 3.

- Parse tree:



L.M.D

R.M.D

* Syntax-Directed translation:

In translating programming language construct, consider following syntax directed defination. A syntax directed defination specifies the translation of a construct in terms of attributes associated with semantic components.

Eg: An infix expression into postfix translation.

(i) If E is a variable or constant, then the postfix notation for E is E itself.

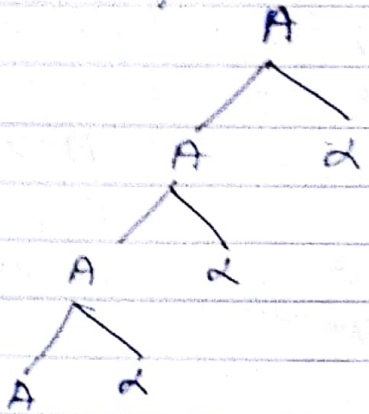
(ii) If E is an expression of the form E_1 op E_2, where op is any binary operator, then the post-fix notation for E is $E_1'E_2'$ op where E_1' and E_2' are the postfix notation for E_1 and E_2 respectively.

(iii) If E is an expression of the form (E_1), then the postfix notation for E_1 is also the postfix notation for E.

* Left-recursive :-

A grammar (syntax defination) is said to be left recursive if any production is of the form $A \rightarrow A\alpha$. The left recursive grammar leads the recursive descent parser into infinite loop so top-down recursive parsing couldn't handle such grammar.

The parse tree will look like,



Removing left recursive, a grammar of the form $A \rightarrow A\alpha / \beta$ where β does not start with A can be written as,

$$A \rightarrow \beta R$$
$$R \rightarrow \alpha R / \epsilon$$

which is the equivalent grammar with left recursive.

After elimination of left recursion,

$$A \rightarrow \beta R$$
$$\rightarrow \beta\alpha R$$
$$\rightarrow \beta\alpha\alpha R$$
$$\rightarrow \beta\alpha\alpha\alpha R$$
$$\rightarrow \beta\alpha\alpha\alpha\alpha \epsilon$$

In general,

Left Recursive grammar :-

$$A \rightarrow A\alpha_1 / A\alpha_2 / \ldots / A\alpha_n / \beta_1 / \beta_2 / \ldots / \beta_m$$

where β does not start with A.

Removing the recursion, the grammar equivalent to the left recursion is

$A \to \beta_1 R \mid \beta_2 R \mid \dots \mid \beta_m R$

$R \to \alpha_1 R \mid \alpha_2 R \mid \dots \mid \alpha_n R \mid \epsilon$

Example :- $E \to E+T \mid T$ (A, α, β)

$T \to T * F \mid F$

$F \to (E) \mid a$.

Sol^n :- $E \to TR$

$R \to +TR \mid \epsilon$

$T \to FR_1$

$R_1 \to *F \mid \epsilon$

$F \to (E) \mid a$

$A \to A\alpha \mid \beta$.

$A \to \beta R$

$R \to \alpha R \mid \epsilon$

Qs> * $X \to Xa \mid Xab \mid Y \mid bc$

$Y \to Yb \mid YA \mid ab$.

$A \to aA \mid \epsilon$.

Ans :- $X \to YR_1 \mid bcR_1$

$R_1 \to aR_1 \mid abR_1 \mid \epsilon$

$Y \to abR_2$

$R_2 \to bR_2 \mid AR_2$

$A \to aA \mid \epsilon$

29th August

* Abstract Stack Machine :-

- The front end of a compiler constructs the intermediate representation of source code program from which the back-end generates the target program. One popular intermediate representation is

code for an abstract stack machine.

- The machine has separate instruction, data memories and all arithmetic operations are performed on stack.
- The instructions are quite limited and fall into three classes:

a) Arithmetic Instructions.
b) Stack Manipulation.
c) Control Flow.

a) Arithmetic Instructions:-

- The abstract machine must implement each operator in the intermediate language.
- A basic operation such as addition, substraction, multiplication is supported directly by the abstract machine.
- The abstract machine code for an arithmetic expression simulates the evaluation of a postfix representation for the expression using stack.

For example: In the evaluation of the postfix expression 25+6* the following actions are performed.

1. Push 2 in the stack.
2. Push 5 in the stack.
3. Pop them and add the two elements and push the result 7 in stack.
4. Push 6 in the stack.
5. Pop them and multiply the two elements and push the result 42 in stack.

- L-values and r-values :-
  - L-values of an identifier refers to the location for the value to be stored.
  - r-value of an identifier is the actual value of it.

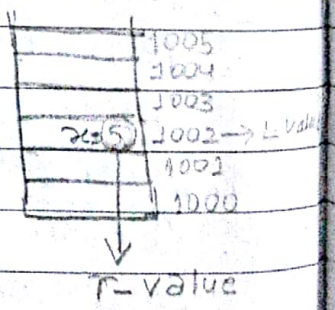
  Eg: int x = 5

  Here, x is an identifier that is placed in the memory location which is L-value and the constant 5 is stored in that location which is R-value.

b) Stack Manipulation :-

The instructions to access data memory are:

i) push v :- push v onto the stack
ii) rvalue l :- push contents of data to location l.
iii) l value l :- push address of data to location l.
iv) pop :- remove the top element from the stack.
v) copy :- push a copy of the top value on the stack.
vi) + :- add.
vii) * :- multiply.
viii) = :- The r-value is placed in the l-value and both are poped.



c) Control Flow :-

- It helps us to understand the structure of control flow graphs (CFG)
- To determine the loop structure of CFG.
- Finding control dependence requires an CFG.

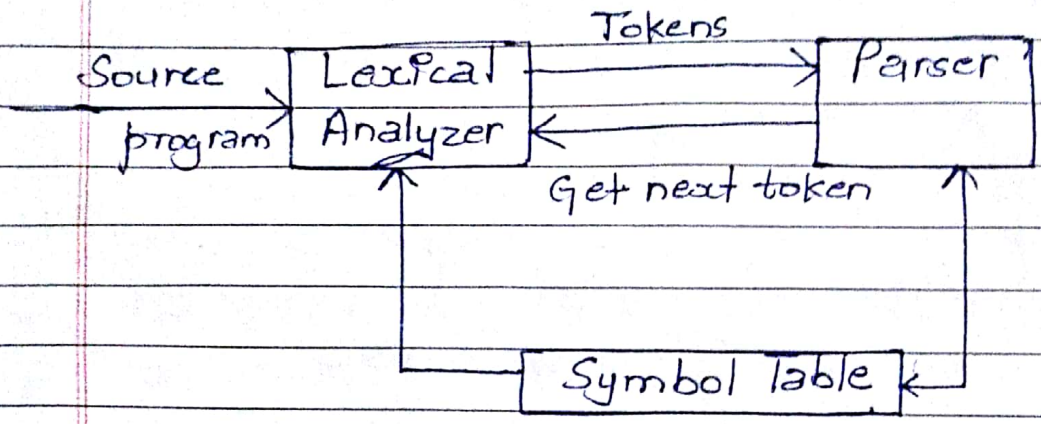
4th September

4th September

Unit: 2

* Role of Lexical Analysis :-

The lexical analyzer is the first phase of a compiler. It's main task is to read the input characters and produce an output as a sequence of tokens that the parser uses for syntax analysis.



Upon receiving a 'get next token' command from the parser, the lexical analyzer reads the input character until it can identify the next token. The LA returns to the parser representation for the token it has found. It also perform other auxiliary operation like:

a) Removing Redundant whitespaces.
b) Removing the comments.
c) Identifying the tokens by matching pattern
d) Stores into data structure called symbol table.

a) Token :-

Any logical building block of a language. Typical tokens are:

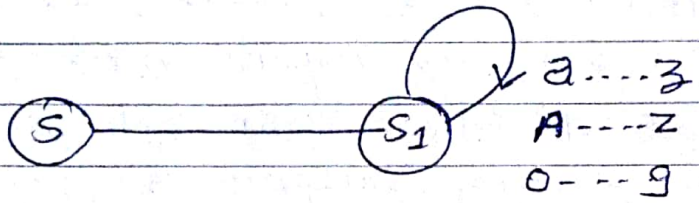
1) Identifier
2) Keyword.
3) Operator
4) Special symbol.

b) Pattern :-

The set of strings is described by a rule called pattern associated with token.

Eg: letter (letter/digit)*

· is a pattern to symbolized the set of string which consists of letter followed by a letter or digit.



c) Lexeme :-

A lexeme is a sequence of character that is matched by the pattern of a token.

Eg: DIV = 6/5;

Lexeme	Token
DIV	Identifier
=	Aggi Assignment symbol.
6	num
/	division op.
5	num
;	End of statement.

Lookahead and Buffering :-

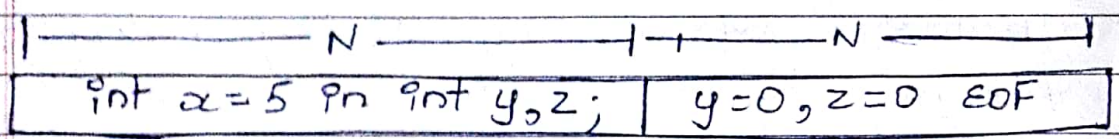
V.V.Imp
T.U. 2068

Many times a scanner has to lookahead character from the current character in order to recognize the token.

For eg :- 'Int' is a keyword in C, while the term int may be an identifier, when

the character 'i' is encountered the scanner cannot decide whether it is a keyword or an identifier until it reads two more characters. In order to move efficiently back and front in the input stream Input buffering is used.

- Buffering technique have been developed to reduce the amount of overhead required to process an input character. We use a buffer divided into 2N character when half of the buffer is represented by N character. This technique is called 2N buffering technique.



Here, N is the no. of characters in a disk block. N input character into each half of the buffer is read by one system command EOF ~~eom~~ character is read in buffer in a case of fewer than N characters remaining in the input. Two pointers are maintained in the input buffer,

1. Lexeme pointer.
2. Forward pointer.

Initially, both pointers point to the first character of the next lexeme to be found. The forward pointer scans ahead until a match for a pattern of lexeme is found. Once the next lexeme is determined, the forward pointer is set to the

character at its right end. If the forward pointer is about to move past the halfway mark, the right half is filled with N new input characters. If the forward pointer is about to move past the right end of the buffer, the left half is filled with N characters. If the eof character is encountered it indicates that the whole input has been scanned

* Specification of tokens:-

The pattern of a token is specified by using regular expression.

V.V.Imp
RU. 2069

- **R.E:-** A regular expression is built up out of simpler regular expression using a set of defining rules. Each regular expression r denotes a language L(r).

The rule that defines the regular expression over alphabet ϵ.

1. ϵ is a R.E that denotes $\{\epsilon\}$ i.e. the set containing the empty string.
2. If 'a' is a symbol in E, then 'a' is a regular expression that denotes $\{a\}$ i.e. the set containing the string 'a'
3. Suppose r and s are R.E. denoting the language L(r) and L(s) then,

a) (r)|(s) is R.E denoting $L(r) \cup L(s)$.
b) (r).(s) is R.E denoting $L(r).L(s)$.
c) (r)* is R.E. denoting $(L(r))^*$.
d) (r) is R.E. denoting L(r).

Example: Let $\Sigma = \{a, b\}$

- The R.E. a/b denotes the set $\{a, b\}$
- The R.E. (a/b)(a/b) denotes $\{aa, ba, ab, bb\}$
- The R.E. a^* denotes the set of all strings of zero or more a's i.e. $\{\epsilon, a, aa, aaa, \ldots\}$

- Properties of R.E:-

If r, s and t are R.E;

1) $r+s = s+r$ → commutative rule for union.
2) $r(s+t) = rs + rt$ → left distributive.
3) $(r+s)t = rt + st$ → right distributive.
4) $(r+s)+t = r+(s+t)$ } → union and concatenation
5) $(r.s).t = r.(s.t)$ } operations are associative
5) Identity rule → $\epsilon.r = r.\epsilon = r$ → Identity rule for concatenation.

$\phi + r = r + \phi = r$ → Identity rule for union.

6) Annihilator → ϕ is an annihilator for concatenation. i.e. $\phi.r = r.\phi = \phi$
7) Rules for closure → $(r^*)^* = r^*$

$r^* = r.r^* = r^*.r$

* Recognition of tokens:

The recognition of tokens implies the implementation of R.E. recognize i.e. a finite state machine.

- Finite automation:- A regular expression are recognized by constructing a generalized transition diagram called finite automata. A finite automata can be deterministic or non deterministic DFA is faster recognizer of R.E. than NFA.

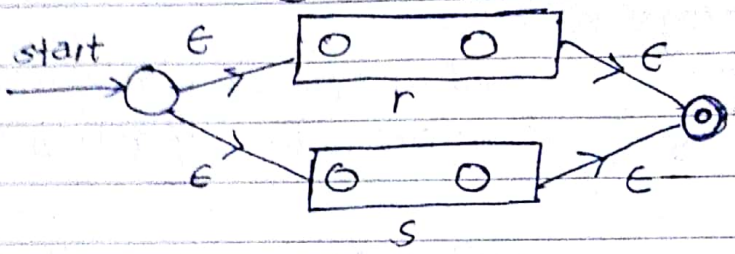
* Thomson's construction for FA :- From R.E to ε-NFA

1) ∅ : start → ●

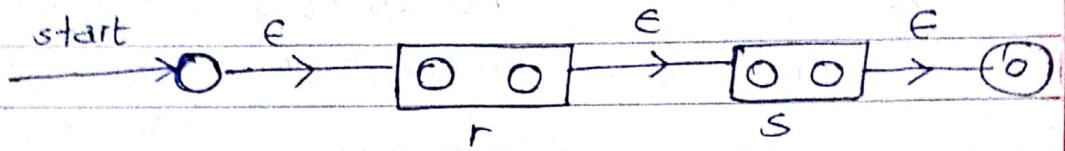
2) ε :- start → ε → ◎

3) a :- start → a → ◎

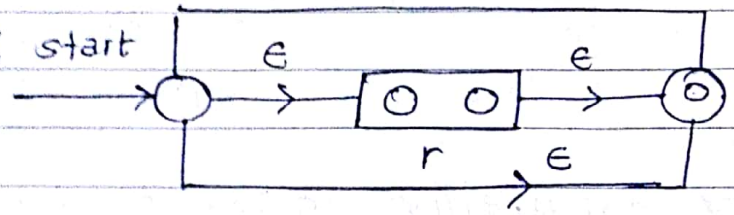
4) r+s :-



5) rs :-



6) r* :



Eg :- 01 + 10, (01+00)*

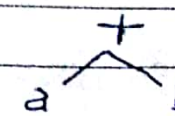
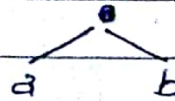
* Conversion of R.E. to DFA :-

1) Augmented Regular Expression :- add # at the end of R.E. Example : (a+b)* abb ⇒ (a+b)* abb#

2) Construct syntax free of R.E :
- Each interior nodes of syntax free operator and leaf nodes represent a terminal or #.

3) Compute the firstpos (i), lastpos (i) and followpos(i) from the syntax free.

4) Compute the DFA using follows:
Rules for computing nullable (), firstpos () and lastpos ().

n	nullable()	firstpos()	lastpos()
1> n is left node containing null.	true	{ }	{ }
2> non-leaf node ~~containing null~~	false	{i}	{i}
3> 	nullable(a) or nullable(b)	firstpos(a) U firstpos(b)	lastpos(a) U lastpos(b)
4> 	nullable(a) and nullable(b)	if nullable(a) then firstpos(a) U firstpos(b) else firstpos(a)	if nullable(b) then lastpos(a) U lastpost(b) else lastpos(b)
5> 	true	firstpos(a)	lastpos(a)

9th September

* Rules for calculating followpos() :-

(1) If n is a cat-node i.e. $n = a \cdot b$ and 'i' is in lastpos(a) then all the position in firstpos(b) are in followpos(i)

$$followpos(i) = firstpos(b)$$

$$i \in lastpos(a)$$

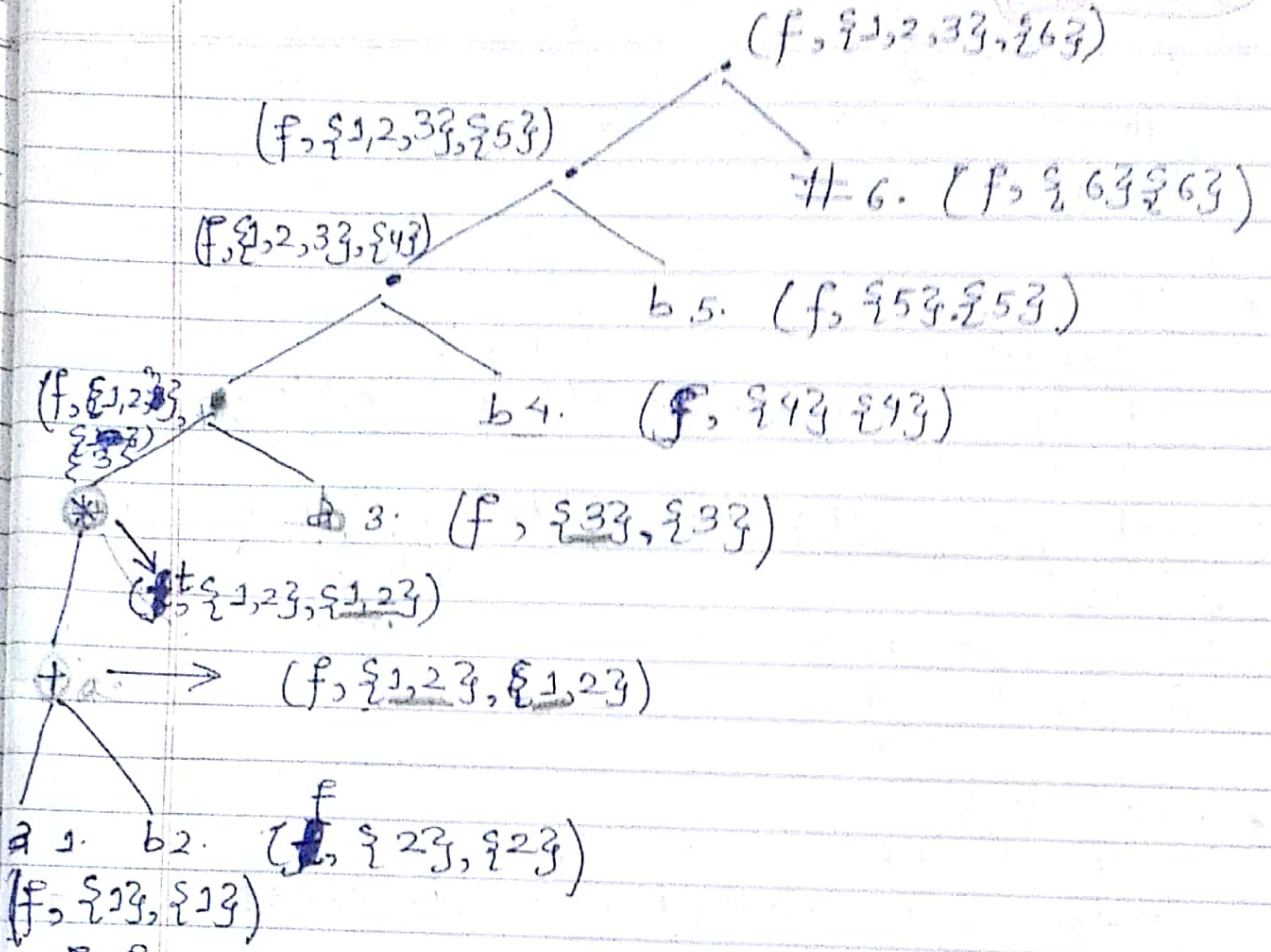
(2) If n is starnode i.e. $n = a^*$ and i is a position in lastpos(a) then all positions in firstpos(a) are followpos(i)

$$followpos(i) = firstpos(a)$$

$$i \in lastpos(a)$$

Example :-

Given R.E is $(a+b)^*abb$ convert it into DFA.



Computing followpos :

i) followpos(1) = {3, 1, 2} = {1, 2, 3}
ii) followpos (2) = {3, 1, 2} = {1, 2, 3}
iii) followpos (3) = {4}
iv) followpos (4) = {5}
v) followpos (5) = {6}
vi) followpos (6) = {∅}

DFA construction :-

i) Start node = first position of root = {1, 2, 3} = S_1
∴ S = {S_1}

ii) Mark S_1

position 1, 3 = a , 2 = b.

followpos (1) ∪ followpos (3) = {1, 2, 3, 4} = S_2
∴ S = {S_1, S_2}

$\delta(S_1, a) = S_2$

followpos(2) = {1, 2, 3} = S_1

$\delta(S_1, b) = S_1$

- Mark S_2 :

position : 1, 3 = a ; 2, 4 = b.

followpos(1) ∪ followpos(3) = {1, 2, 3, 4} = S_2

$\delta(S_2, a) = S_2$

followpos(2) ∪ followpos(4) = {1, 2, 3} ∪ {5}

= {1, 2, 3, 5} = S_3

∴ S = { S_1, S_2, S_3 }

$\delta(S_2, b) = S_3$

- Mark S_3 :

position : 1, 3 = a ; 2, 5 = b.

followpos(1) ∪ followpos(3) = {1, 2, 3, 4} = S_2

$\delta(S_3, a) = S_2$

followpos(2) ∪ followpos(5) = {1, 2, 3, 6}

= S_4

∴ S = { S_1, S_2, S_3, S_4 }

$\delta(S_3, b) = S_4$

- Mark S_4 :

position : 1, 3 = a ; 2 = b.

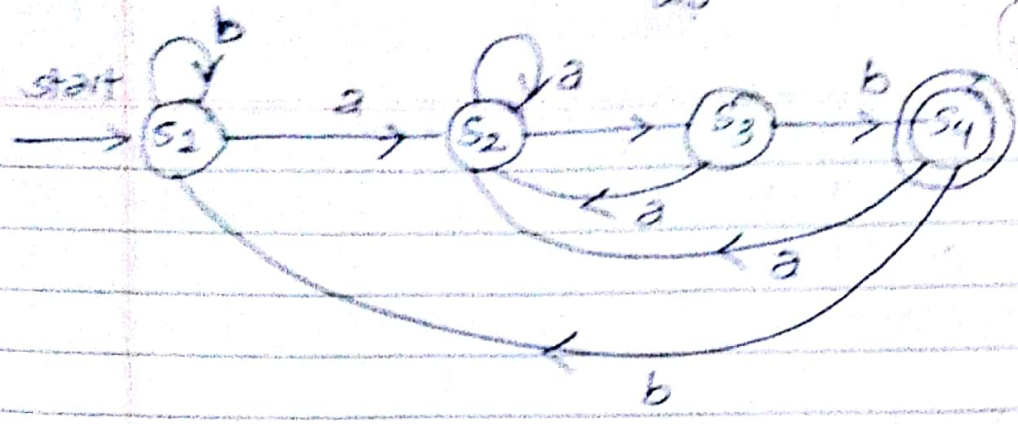
followpos(1) ∪ followpos(3) = {1, 2, 3, 4} = S_2

$\delta(S_4, a) = S_2$

followpos(2) = {1, 2, 3} = S_1

$\delta(S_4, b) = S_1$

So, final DFA is :



* Assignment: 1

Q> Convert R.E. into DFA

1. $(a+b)^* abb^* a$
2. $(a+\epsilon)^* bb^*$
3. $(a/b)^* ab^*$

* Converting ϵ-NFA into NFA :-

- If ϵ-NFA accepts empty string, the first state of NFA from ϵ-NFA should be marked as final state.
- For each state q and for each input a,

$$\delta_N(q,a) = \epsilon\text{-closure}(\delta_\epsilon(\epsilon\text{-closure}(q), a))$$

Given ϵ-NFA:

Sol^n:- Start state of NFA from given ϵ-NFA = q_0
Since, ϵ-NFA accepts ϵ, so q_0 is also final state.

Now,

$$\delta_N(q,a) = \epsilon\text{-closure}(\delta_\epsilon(\epsilon\text{-closure}(q_0), a))$$
$$= \epsilon\text{-closure}(\delta_\epsilon(q_0, q_1, q_2), a))$$
$$= \epsilon\text{-closure}(\delta q_0)$$
$$= \{q_0, q_1, q_2\}$$

$\delta_N(q_0, b) = \epsilon\text{-closure}(\delta_\epsilon(\epsilon\text{-closure}(q_0), b))$
$= \epsilon\text{-closure}(\delta_\epsilon(q_0, q_1, q_2), b))$
$= \epsilon\text{-closure}(q_2)$
$= \{q_2\}$

$\delta_N(q_0, c) = \epsilon\text{-closure}(\delta_\epsilon(\epsilon\text{-closure}(q_0), c))$
$= \epsilon\text{-closure}(\delta_\epsilon(q_0, q_1, q_2), c)$
$= \epsilon\text{-closure}(q_2)$
$= \{q_2\}$

$\delta_N(q_1, a) = \epsilon\text{-closure}(\delta_\epsilon(\epsilon\text{-closure}(q_1), a))$
$= \epsilon\text{-closure}(\delta_\epsilon(q_1, q_2), a)$
$= \epsilon\text{-closure}(\phi)$
$= \{\phi\}$

$\delta_N(q_1, b) = \epsilon\text{-closure}(\delta_\epsilon(\epsilon\text{-closure}(q_1), b))$
$= \epsilon\text{-closure}(\delta_\epsilon(q_1, q_2), b))$
$= \epsilon\text{-closure}(q_2)$
$= \{q_2\}$

$\delta_N(q_1, c) = \epsilon\text{-closure}(\delta_\epsilon(\epsilon\text{-closure}(q_1), c)$
$= \epsilon\text{-closure}(\delta_\epsilon(q_1, q_2), c)$
$= \epsilon\text{-closure}(q_2)$
$= \{q_2\}$

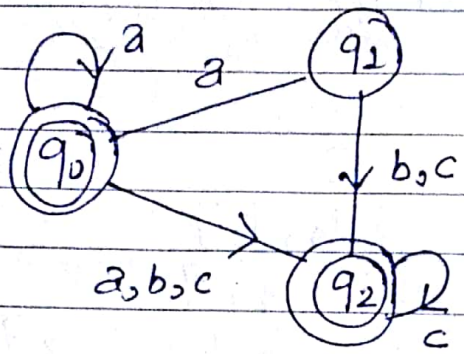
$\delta_N(q_2, a) = \epsilon\text{-closure}(\delta_\epsilon(\epsilon\text{-closure}(q_2), a)$
$= \epsilon\text{-closure}(\delta_\epsilon(q_2, a))$
$= \epsilon\text{-closure}(\phi)$
$= \{\phi\}$

$\delta_N(q_2, b) = \epsilon\text{-closure}(\delta_\epsilon(\epsilon\text{-closure}(q_2), b)$
$= \epsilon\text{-closure}(\delta_\epsilon(q_2, b))$

= ε-closure (ϕ)
= {ϕ}

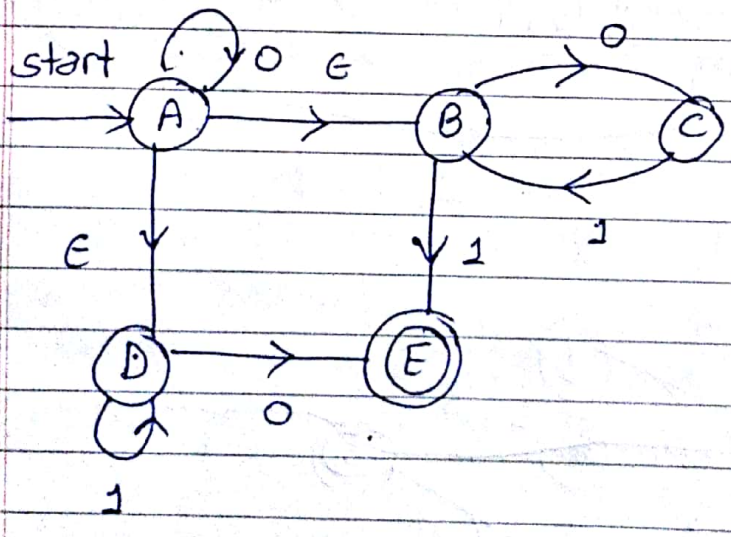
$\delta_N(q_2, c) = \epsilon\text{-closure}(\delta_\epsilon(\epsilon\text{-closure}(q_2), c))$
$= \epsilon\text{-closure}(\delta_\epsilon(q_2, c))$
$= \epsilon\text{-closure}(q_2)$
$= \{q_2\}$

Final NFA is:



Assignment : 2

Qs) Convert into NFA:



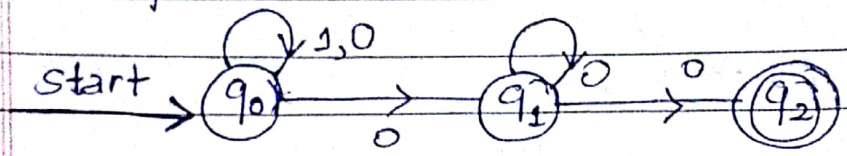
10th September

Tu. Vo. Imp
T.U. 2068

* Convertion of NFA into DFA (subset construction):

The subset construction start from a NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. The goal is to construct equivalent DFA, $D = (Q_D, \Sigma, \delta_D, q_0, F_D)$ such that $L(N) = L(D)$.

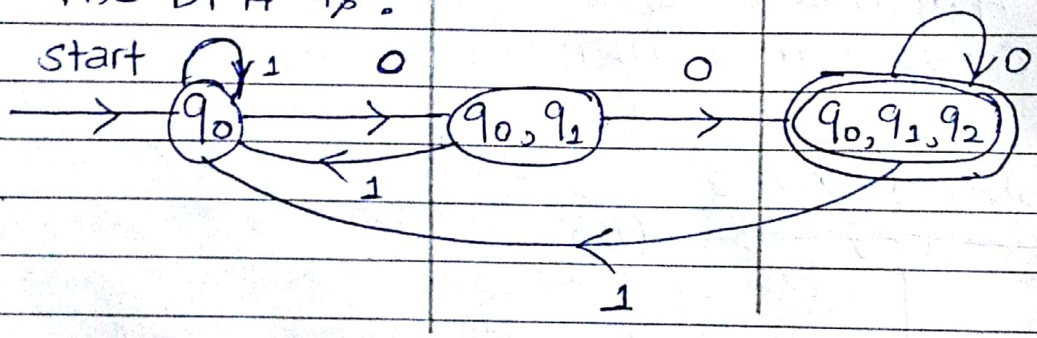
- Example: Convert the following NFA into DFA.



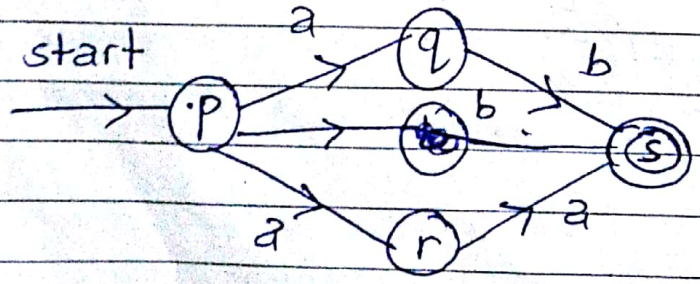
Solⁿ:-

Subsets	0	1
→ {q0}	{q0, q1}	{q0}
{q0, q1}	{q0, q1, q2}	{q0,}
{q0, q1, q2} *	{q0, q1, q2}	{q0}

The DFA is:

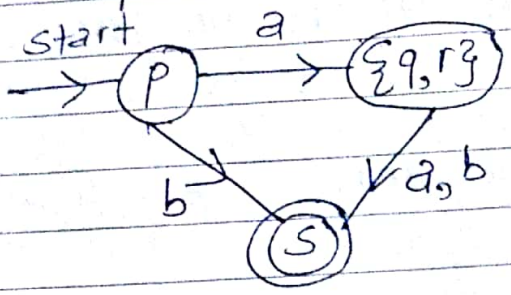


Qs>2.



Sol^n:- Subsets	a	b
→{p}	{q,r}	{s}
{q,r}	{s}	{s}
*{s}	∅	∅

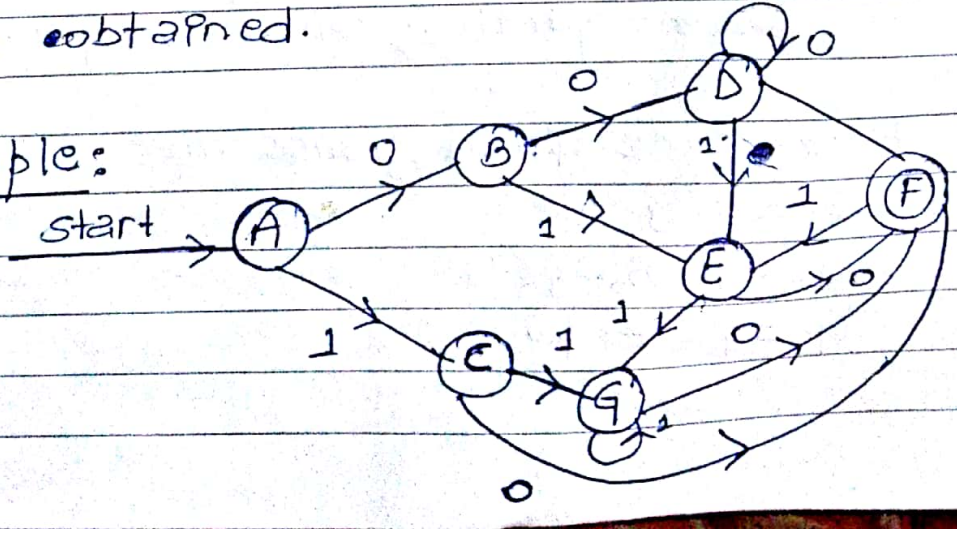
The equivalent DFA is:



* Minimization of DFA :-
- Identify the equivalent and non-equivalent states.
- Combine those equivalent states into a single state.
- Initially, from the two groups of states, one for final state and next for the non-final states.
- Breakdown the group containing more than one states by checking transitions by each input symbol.
- continue until, there are no separation of groups obtained.

Example:

1.



Sol^n:- Here, final state = F and other non-final state

So, initial distinguishable groups

$G_1 : \{F\}$

$G_2 : \{A, B, C, D, E, G\}$

Now, take group G_2,

	A	B	C	D	E	G
0-leads:	G_2	G_2	G_1	G_2	G_1	G_1
1-leads:	G_2	G_2	G_2	G_2	G_2	G_2

So, the new groups are:

$G_1 = \{F\}$

$G_2 = \{A, B, D\}$

$G_3 = \{C, E, G\}$

Again, take groups G_2,

	A	B	D
0-leads	G_2	G_2	G_2
1-leads	G_3	G_3	G_3

So, no further separation found.

Now, take group G_3,

	C	E	G
0-leads	G_1	G_1	G_1
1-leads	G_3	G_3	G_3

So, no further separation found.

Hence, the final groups are:

$G_1 = \{F\}$

$G_2 = \{A, B, D\}$

$G_3 = \{C, E, G\}$

Sol^n:- Here, final state = F and other non-final state
So, initial distinguishable groups

$G_1 : \{F\}$

$G_2 : \{A, B, C, D, E, G\}$

Now, take group G_2,

	A	B	C	D	E	G
0-leads:	G_2	G_2	G_1	G_2	G_1	G_1
1-leads:	G_2	G_2	G_2	G_2	G_2	G_2

So, the new groups are:

$G_1 = \{F\}$

$G_2 = \{A, B, D\}$

$G_3 = \{C, E, G\}$

Again, take groups G_2,

	A	B	D
0-leads	G_2	G_2	G_2
1-leads	G_3	G_3	G_3

So, no further separation found.

Now, take group G_3,

	C	E	G
0-leads	G_1	G_1	G_1
1-leads	G_3	G_3	G_3

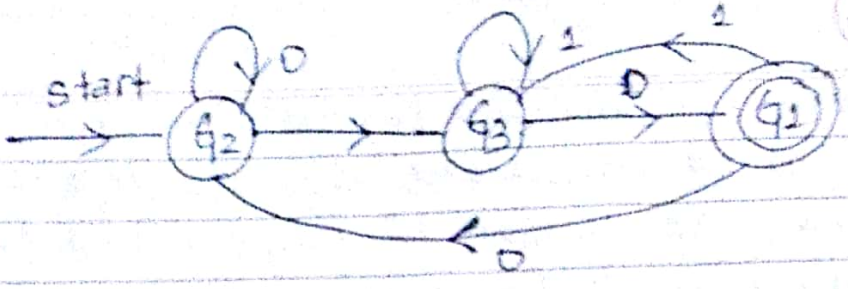
So, no further separation found.

Hence, the final groups are:

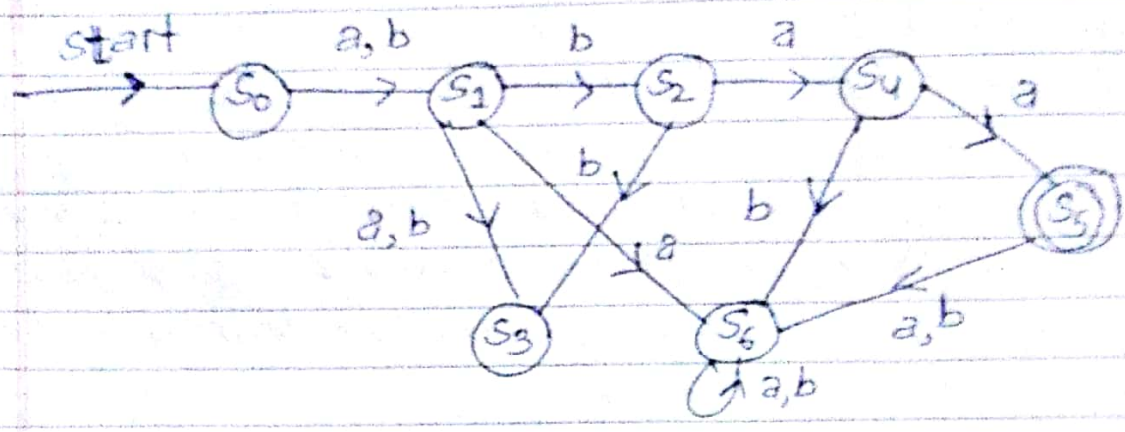
$G_1 = \{F\}$

$G_2 = \{A, B, D\}$

$G_3 = \{C, E, G\}$



2.

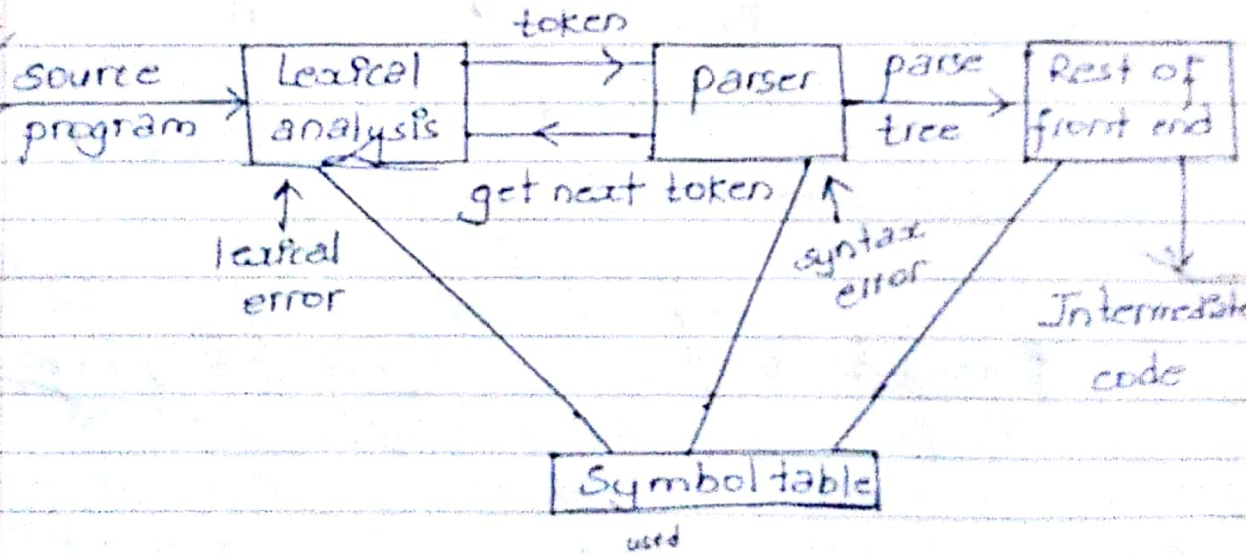


Unit - 2

2.2

* Syntax analysis:

The role of the parser:



A syntax analyzer is to analyze the source program based on the definition of its syntax. It works in lookup step with the lexical analyzer and is responsible to create parse tree from the source code. A parser increments a context

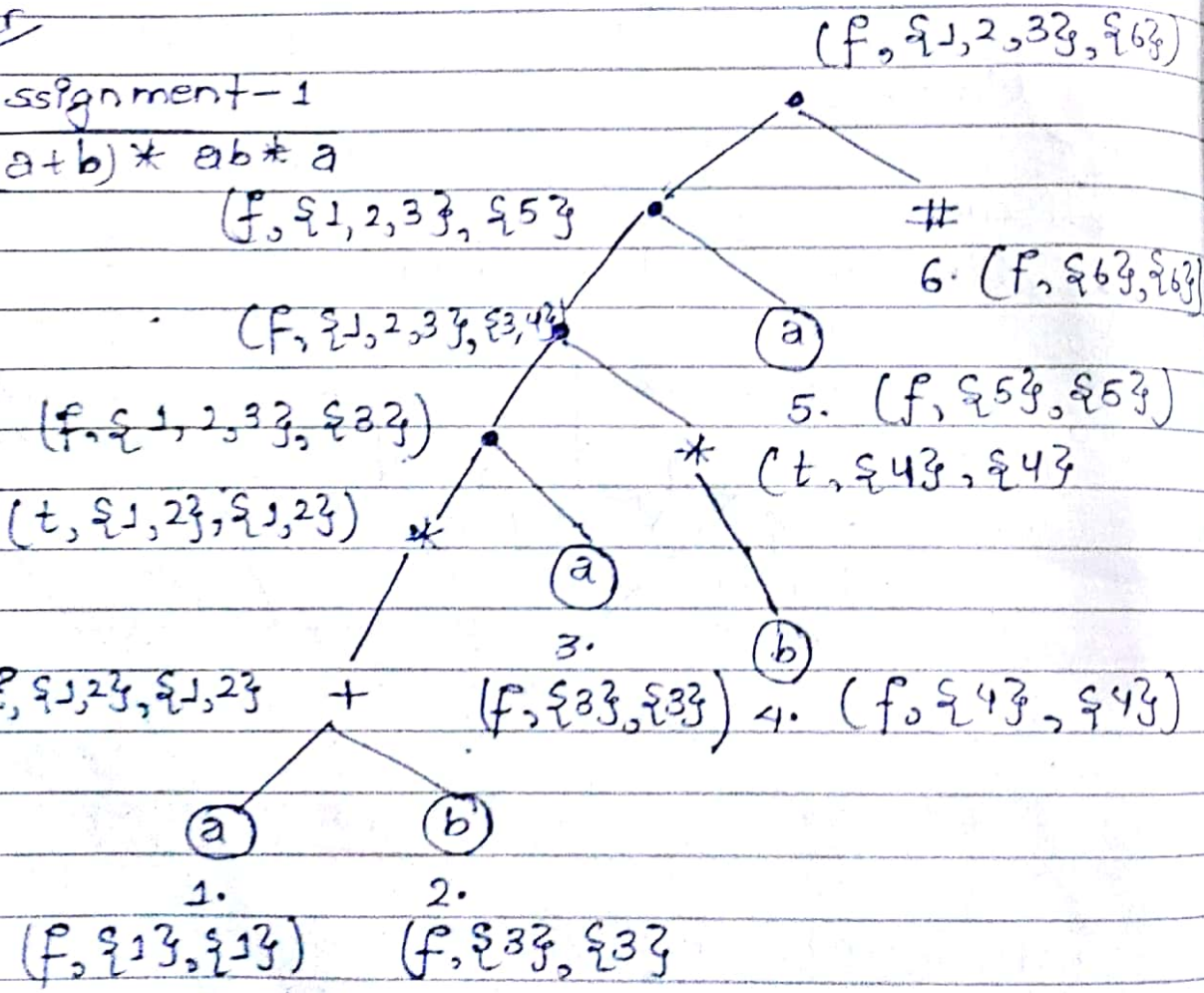
free grammar whereas lexical analyzer implements a finite automata. Besides the checking of syntax, the parser is responsible to report the syntax error in the source program. A parser is also responsible to invoke semantic actions.

- for static semantic checking. For eg: type checking of expression function.
- for syntax directed translation of source code into an intermediate representation. Eg: control flow graph, 3-address code.

11th September

* Assignment-1

(1) (a+b)* ab* a



Now, calculating followpos,

i) followpos(1) = {3, 1, 2} = {1, 2, 3}

ii) followpos(2) = {1, 2, 3}

iii) followpos(3) = {5,4} = {4,5}
iv) followpos(4) = {5,4} = {4,5}
v) followpos(5) = {6}
vi) followpos(6) = {∅}

Then,
constructing DFA,
Start node = firstpos of root node = {1,2,3} = S_1.
∴ S = {S_1}

- Mark S_1 :
  position : 1,3 = a ; 2 = b.

→ followpos of (1) ∪ followpos(3) = {1,2,3,4,5} = S_2
δ(S_1, a) = S_2

→ followpos(2) = {1,2,3} = S_1
δ(S_1, b) = S_1

- Mark S_2 :
  position: 1,3,5 = a ; 2,4 = b.

→ followpos(1) ∪ followpos(3) ∪ followpos(5) = {1,2,3,4,5,6}
→ followpos(2) ∪ followpos(4) = S_3
- = {1,2,3,4,5} = S_2
δ(S_2, a) = S_3
δ(S_2, b) = S_2
∴ S = {S_1, S_2, S_3}

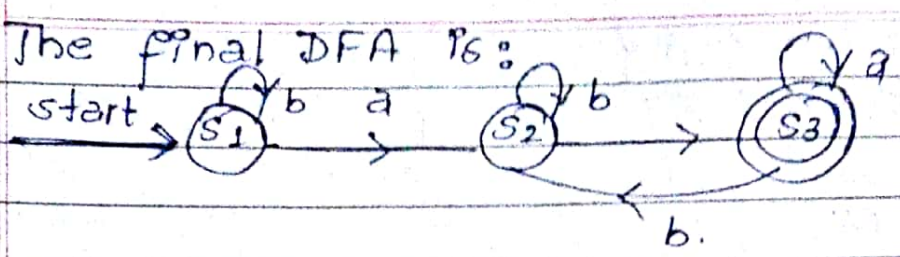
- Mark S_3 :
  position : 1,3,5 = a ; 2,4 = b.

→ followpos(1) ∪ followpos(3) ∪ followpos(5) = {1,2,3,4,5,6}
S_3, a = S_3
∴ δ(~~a~~, S_3) = S_3

→ followpos(2) ∪ followpos(4) = {1,2,3,4,5} = S_2
∴ δ(~~S_3~~ S_3, b) = S_2

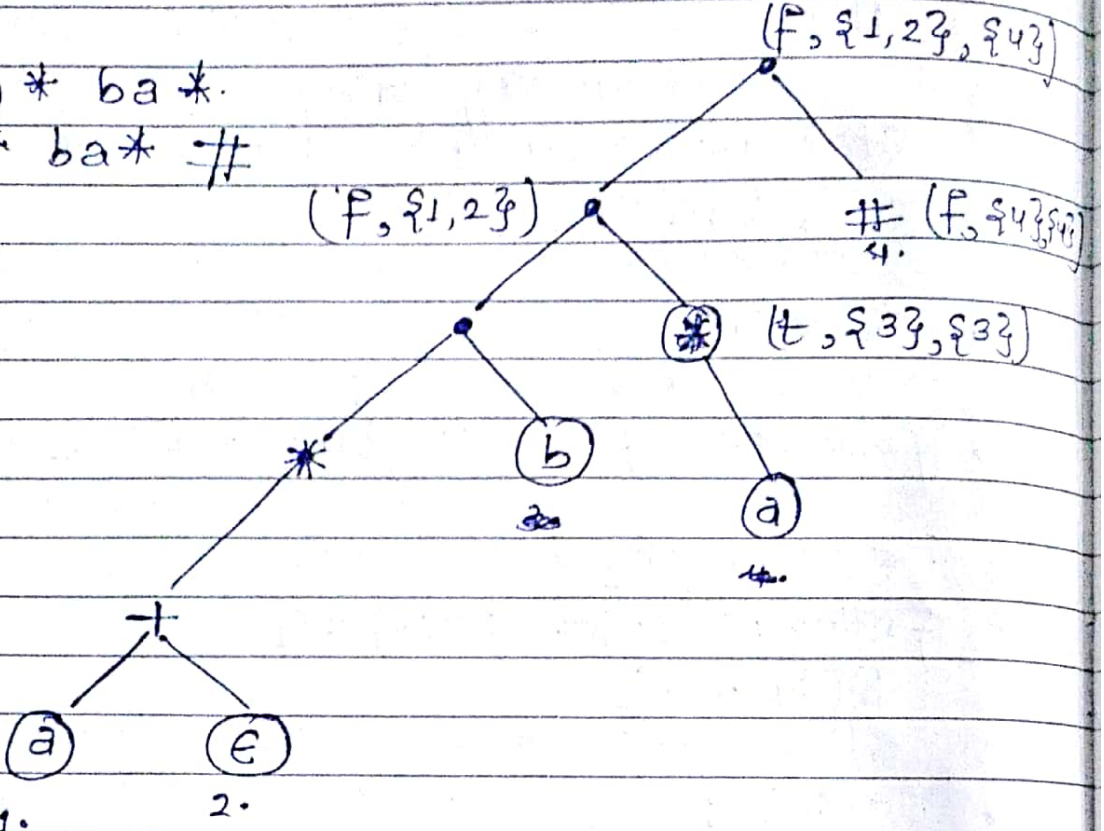
∴ δ = {S_1, S_2, S_3}

The final DFA is:



b) $(a+\epsilon)^* ba^*$

$\Rightarrow (a+\epsilon)^* ba^* \#$



followpos(1) = {2,1} = {1,2}
followpos(2) = {4,3} = {3,4}
followpos(3) = {3,4}
followpos(4) = {∅}

Now, DFA construction,

· start state = root_node (firstpos) = {1,2} = S_1
∴ $S = \{S_1\}$

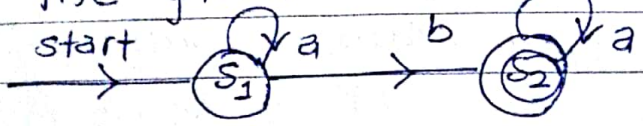
- Mark S_1

position: 1 = a ; 2 = b
followpos(1) = {1,2} = S_1
$\delta(S_1, a) = S_1$

followpos (2) = {3, 4} = S_2
∴ $\delta(S_1, b) = S_2$

- Mark S_2 :
position : 3 = a
followpos (3) = {3, 4} = S_2
∴ $\delta(S_2, a) = S_2$

The final DFA is:



* Syntax error handling :-
A good compiler should assist in identifying and locating errors. Program may contain errors at different levels.

(i) Lexical error :-
Misspelling on identifier, keyword, or operator compiler can easily recovered and continue from error.
Example : int num;
nu = 5;

(ii) Syntax error :-
An expression with unbalanced parentheses or operators misplace etc. Such error are most important and almost always recovered during compilation.
Example : while (1 { printf ("hi "); }

(iii) Semantic error :-
These are important which are either static or dynamic. Static semantic errors can

be sometimes recovered but dynamic semantic errors are very hard and cannot be recovered during compilation.
A compiler cannot easily identify such type of error.
Eg :- Int main() { return 'c'; }
Eg :- int calculateArea (int width, int height)
{
return width + height
}

(iv) Logical error :-
These are very hard for detect and recovered at compile time which are impossible to recover by compiler. For eg :- use of wrong operators.
Example : if (a=1) { printf ("hi"); }

16th September

* Parsing :-
Parsing can be either top-down or bottom-up. Top-down parsing check if a string generated by a grammar is starting from the initial symbol and ~~always~~ working down. Whereas bottom-up parser check if a string generated by grammar is starting from the leaves and working up.

- Recursive Descent Parsing :-
(A top down parsing method)

– In recursive descent parsing, parsing starts from the start variable (root).

- It first replaces the variable at each step by the first production of the variable.
- If choosing first production fails to derive the terminal string, the derivation is back-tracked and uses the next production and so on.

Example: consider a CFG.

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

and the input string $w = cad$.

Consider the parser string α = S (start symbol)

1) Let iptr be index of input string and optr the index of output string.

Initially, iptr = optr = 0

input	output
iptr (cad)	optr (S)

2) while α[optr] is non-terminal, expand the non-terminal by its production.

input	output
iptr (cad)	optr (cAd)

3) while w(iptr) = α (optr) then advance both iptr and optr if end of string reached parsing successful.

input	output
c[iptr (ad)]	c[optr (Ad)].
c[iptr (ad)]	c[optr (abd)] → Rule 2
ca[iptr (d)]	ca[optr (bd)].

4) The while loop above stops if:

a) a non-terminal is encountered in α.
b) end of string is reached.
c) If w [iptr] ≠ α [optr].

5> If (a) is true, the goto step 2 and expand non-terminal with the first production.
If (b) is true, then exit with success.
If (c) is true then, revert iptr and optr to the place they were in last expansion and replace the non-terminal with its next production and go ahead.

Now, backtracking upto the last expansion,

Input	output
c[iptr(ad)]	c [optr(Ad)]
c[iptr(ad)]	c [optr(ad)] → Replace A with A→a
ca[iptr(ad)]	ca [optr(d)] → Rule 3.
cad [iptr()]	cad [optr()] → Rule 3.

Here, 4(b) is true so the parsing is successful.

Example: CFG:

R → id s | (R) | s
S → +RS | ·RS | *S | ∈
id → a | b.

Determine ~~each~~ whether the following string are the language of grammar tracking the recursive descent algorithm.

(i) (b.a.b)* (ii) a.b.b.a.

Sol:- Here, $\alpha = R$ (start symbol)

Rule:1 Initially, iptr = optr = 0

input	output
iptr(a.b.b.a)	optr(R)

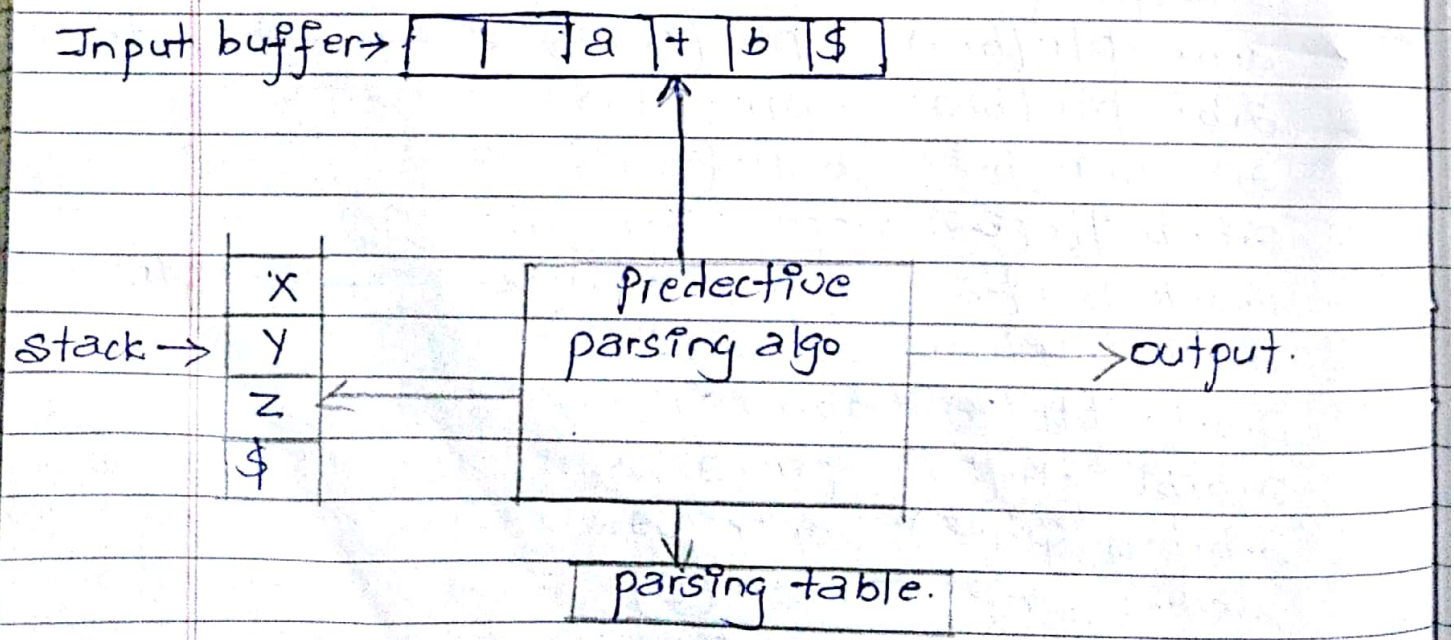
Rule:2

input	output	
iptr(a.b.b.a)	optr(Rids)	
iptr(a.b.b.a)	optr(Rs)	
a iptr(.b.b.a)	optr(s)	← Backtrack.
a iptr(.b.b.a)	optr(+RS)	← backtrack
a iptr(.b.b.a)	optr(s)	
a iptr(.b.b.a)	optr(.Rs)	
a.iptr(.b.b.a)	optr(.idss)	
a.iptr(b.b.a)	optr(.ass)	← backtrack
a.iptr(b.b.a)	optr(idss)	
a.iptr(b.b.a)	optr(bss)	
a.b iptr(.b.a)	optr(ss)	
a.b iptr(.b.a)	optr(+Rss)	← backtrack
a.b iptr(.b.a)	optr(ss)	
a.b iptr(.b.a)	optr(.Rss)	
a.b.iptr(b.a)	optr(idsss)	
a.b.iptr(b.a)	optr(asss)	← backtrack
a.b.iptr(b.a)	optr(idss)	
a.b.iptr(b.a)	optr(bsss)	
a.b.b iptr(.a)	optr(sss)	
a.b.b iptr(.a)	optr(+Rsss)	← backtrack
a.b.b iptr(.a)	optr(.Rsss)	
a.b.b.iptr(a)	optr(idsss)	
a.b.b.iptr(a)	optr(assss)	
a.b.b.a iptr()	optr(ssss)	
a.b.b.a iptr()	optr(ϵ)	
a.b.b.a iptr()	optr()	

Hence, parsing successful.

- This algorithm works but not suitable for the large ~~amount~~ grammar due to its backtracking action.
- This is very slow.
- If the grammar is left recursive, then this algorithm does not work.
- The recursive descent parser always choose the first production for a variable to be replaced which leads too much backtracking action and also leads to infinite recursion for left recursion grammar but predictive parser predicts for derivation from which the input symbol best match with the production.

- The non-recursive predictive parser:
  The non-recursive predictive parser includes an input-buffer, a stack, a parsing table and output string. It is table driven parsing method.



18th September

4 * Non-Recursive predictive parsing:

imp

WU 2065

The input buffer and stack are delimited by a special symbol '$' that denotes the end of the stack.

2) The parsing table is made of entries of the form $m[X,a] = \alpha$ which says that if the stack points to non-terminal 'X' and input buffer points to 'a' then 'X' has to be replaced by α. α may be a set of non-terminals and terminals or an errors.

3) The program for parser behaves as follows:

a) If $X = a = \$$, the parser halts and announce the successful parsing.

b) If $X = a \neq \$$, the parser pops X off the stack and advance the input pointer to the next input symbol.

c) If X is non-terminal, the program consults the entry $m[X,a]$ of parsing table m. This entry will be either production of X or error. If it is X production like $X \rightarrow uvw$ it replace the top of stack (X) by wvu 'u' becoming the new top element, if $m(X,a)$ is error then the parser calls an error recovery routine.

- Algorithm for non-recursive predictive parsing :-

i) Input : A string 'w' and a parsing table m for CFG.

ii) Method : Initially the parser is in configuration $s on the stack with s on the top and w$ in input buffer.

Step:1 Set the input pointer (ip) to the first symbol of w$.

Step:2 Set stack to $ S where S is start variable of CFG.

Step:3 a) Let X be the top of stack symbol and 'a' is the current symbol pointed by ip.

b) If X is a terminal or $

(i) If X = a then pop X from stack and advance ip to next input symbol.

else error.

else

/* X is a non-terminal */

(ii) If m[X, a], $X \to Y_1, Y_2, Y_3, \ldots Y_k$ then

- pop X from stack.
- push $Y_k, Y_{k-1}, \ldots, Y_2, Y_1$ onto stack.
- output production $X \to Y_1, Y_2, Y_3, \ldots Y_k$

else error.

Step:4 until X = $

Example :-

1) Consider the following grammar

$E \to E+T / T$

$T \to T*F / F$

$F \to (E) / id$

Input string $w = id + id * id$

Sol^n:- Here,

Eliminating the immediate left recursion,

$E \to TR$

$R \to +TR / \epsilon$

$T \to FR_1$

$R_1 \to *FR_1 / \epsilon$

$F \to (E) / id.$

$E \to TE'$

$E' \to +TE' / \epsilon$

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to (E) / id$

Now, the predictive move of parser are,

Stack	input	output
$E	id+id*id$	—
$E'T	id+id*id$	E → TE' ← stack
$E'T'F	id+id*id$	T → FT'
$E'T'id	id+id*id$	F → id
$E'T'	+id*id$	—
$E'	+id*id$	T' → ϵ
$E'T+	+id*id$	E' → +TE'
$E'T	id*id$	—
$E'T'F	id*id$	T → FT'
$E'T'id	id*id$	F → id
$E'T'	*id$	—
$E'T'F*	*id$	T' → *FT'
$E'T'F	id$	—
$E'T'id	id$	F → id
$E'T'	$	T' → ϵ
$E'	$	E' → ϵ
$	$	

Hence, parsing successful.

2> Input string w = id* id+ id.

Sol^n: Here, the predictive move of parser are:

Stack	input	output
$E	id*id+id$	—
$E'T	id*id+id$	E → TE'
$E'T'F	id*id+id$	T → FT'
$E'T'id	id*id+id$	F → id
$E'T'	*id+id$	—
$E'T'F*	*id+id$	T' → *FT'
$E'T'F	id+id$	—

Stack	Input	output
$E'T'Id	Id+Id$	F → Id
$E'T'	+Id$	—
$E'	+Id$	T' → ε
$E'T+	+Id$	E' → +TE'
$E'T	Id$	—
$E'T'F	Id$	T → FT'
$E'T'Id	Id$	F → Id
$E'T'	$	—
$E'	$	T' → ε
$	$	E' → ε

19th September

* Parsing Table:

The parsing table 'm' consists of entries of the form m[X, a] = uvw meaning that if the top of the stack hold 'X' and input symbol is 'a' then 'X' should be replaced by uvw. The construction of parsing table is based on two function.

FIRST AND FOLLOW.

* Calculation of FIRST set:-

To compute FIRST(X) for all grammar symbols X_1 apply the following rules until no more terminals or ε can be added to any FIRST set.

1) If X is terminal, then FIRST(X) is {X}.
2) If X → ε is a production, then add ε to the FIRST(X)
3) If X is non-terminal and $X \to Y_1, Y_2, Y_3 \ldots$ is a production then place 'a' in FIRST(X)

if FIRST(Y) 'a' is in FIRST(Y_1). If Y_1 doesn't derive ϵ then we add nothing more to FIRST(X) but if $Y_1 \Rightarrow \epsilon$ then we add FIRST(Y_2) and so on.

- Calculation of Follow set:-

1) Place $ in follow(S), where 'S' is the start symbol.
2) If there is a production $A \to \alpha B \beta$ then, everything in FIRST(β) except for ϵ is placed in Follow(B).
3) If there is a production $A \to \alpha\beta$ or a production $A \to \alpha B \beta$ where FIRST(β) contains ϵ (i.e. $\beta \to \epsilon$) then everything in Follow(A) is in Follow(B)
4) For any non-terminal A, follow(A) is the set of terminal that can appear immediately to the right of non-terminal.
   i.e. If $S \Rightarrow aAbB$ then 'b' is in follow(A)

Example:

	FIRST	Follow
$E \to TE'$	{(, id}	{$,)}
$E' \to +TE' \mid \epsilon$	{+, ε}	{$,)}
$T \to FT'$	{(, id}	{+, $,)}
$T' \to *FT' \mid \epsilon$	{*, ε}	{+, $,)}
$F \to (E) \mid id$	{(, id}	{*, +, $,)}
$S \to ABCDE$	{a, b, c}	{$}
$A \to a \mid \epsilon$	{a, ε}	{b, c}
$B \to b \mid \epsilon$	{b, ε}	{c}
$C \to c$	{c}	{d, e, $}
$D \to d \mid \epsilon$	{d, ε}	{e, $}
$E \to e \mid \epsilon$	{e, ε}	{$}

↗ replacing A by ε so remaining are $S \to BCDE$

	FIRST	Follow
S → ACB/CbB/Ba	{d, h, e, g, a, b}	{$}
A → da/BC	{d, g, h, ε}	{g, h, $}
B → g/ε	{g, ε}	{$, a, h, g}
C → h/ε	{h, ε}	{g, $, b, h}.

* Construction of parsing table:-

1) For each production A → α of the grammar, do step 2) and 3)
2) For each terminal 'a' in FIRST(α), add A → α to m[A, a].
3) If 'ε' is in FIRST(α) add A → α to M[A, a]

For example:-

Consider the grammar,

	FIRST	FOLLOW
E → TE'	{(, id}	{$,)}
E' → +TE'/ε	{+, ε}	{$,)}
T → FT'	{(, id}	{+, $,)}
T' → *FT'/ε	{*, ε}	{+, $,)}
F → (E)/id	{(, id}	{*, +, $,)}

Now, constructing parsing table,

Non-terminal	Terminal					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→+TE'	E'→ε
T	T→FT'			T→ET'		
T'		T'→ε	T'→*FT		T'→ε	T'→ε
F	F→id			F→(E)		

2)

	FIRST	FOLLOW
S → iEtSS'/a.	{i, a}	{$, e}
S' → eS/ε	{e, ε}	{$, e}
E → b	{b}	{t}

LL(1) ① $S \to aAa/\epsilon$, $A \to abS/\epsilon$ ② $S \to AaAb/BbBa$, $A \to \epsilon$, $B \to \epsilon$ ③ $S \to aABb$, $A \to c/\epsilon$, $B \to d/\epsilon$

Now, constructing parsing table,

Non-terminal	a	b	c	d	$
S	$S \to iEtSS'$	$S \to a$			
S'			$S' \to eS$ / $S' \to \epsilon$		$S' \to \epsilon$
E				$E \to b$	

* LL(1) Grammar:-

A grammar where parsing table has no multiply-defined entries is said to be LL(1). The first 'L' corresponds for scanning the input from left to right and second 'L' for producing a leftmost derivation and '1' for using one input symbol of lookahead at each step to make parsing action decisions.

• Properties of LL(1) grammar:-

In any LL(1) grammar if there exists a rule of the form $A \to \alpha/\beta$ where α & β are distinct then,

1) For any terminal a, if $a \in FIRST(\alpha)$ then, $a \in FIRST(\beta)$
2) Either $\alpha \overset{*}{\Rightarrow} \epsilon$ or $\beta \overset{*}{\Rightarrow} \epsilon$ but not both.
3) If β derives ϵ, then α doesn't derive any string beginning with terminal in Follow(A).

The entry for $m[S', e]$ contains both $S' \to eS$ and $S' \to \epsilon$, since, $Follow(S') = \{e, \$\}$. The grammar is ambiguous and the ambuigity is manifeasted by a choice in what production to use when an 'e' is seen. We can resolve the ambiguity if we choose $S' \to eS$. Note that the choice $S' \to \epsilon$ would prevent 'e' from ever being put on the stack or removed from the input and is therefore surely wrong.

26th September

* Bottom-up parsing :-
In Bottom-up parsing, check if a string generated by grammar is starting from leaves and working up.

* Handle :- Any substring which can be replaced by a non-terminal during bottom-up reduction is called handle.

Example: Consider the grammar,

$S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d$

The sentence abbcde can be reduced to (S) by the following steps.

abbcde

→ aAbcde // A→b

→ aAde // A→Abc.

→ aABe

→ S

* Shift reduce parsing :-
A simple bottom-up parsing technique using a stack based implementation is shift reduced parsing. This is a convinient way to implement a shift reduce parser which uses a stack to hold the grammar symbols and an input buffer to hold the ~~grammar~~ input string w. The parsing method can be described as:

1) Initially the stack contains only the symbol $ and the input buffer contains the input string delimited by $ as w$.

2) while (stack ≠ $) do

a) while there is no handle at the top of stack shift input buffer and push the stack symbol into stack.

b) If there is a handle on the top of stack then pop the handle and reduce the handle with non-terminal and push the non-terminal into stack.

Example: Consider the grammar:

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

Input string $w = id + id * id$.

Stack	Input	Actions
$	id + id * id $	shift
$id	+ id * id $	reduce E→id
$E	+ id * id $	shift
$E +	id * id $	shift
$E + id	* id $	reduce E→id
$E + E	* id $	reduce E→E+E
$ E	* id $	shift
$E *	id $	shift
$E * id	$	reduce E→id
$E * E	$	reduce E→E*E
$E	$	Accept

Example: $S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

$w = abbcde$.

Stack	Input	Actions.
$	abbcde$	~~reduce~~ shift
$a	bbcde$	~~reduce A→b~~ shift
$ab	bcde$	reduce A→b
$aA	bcde$	shift
$aAb	cde$	~~reduce A→b~~ shift c
$aAbc	de$	reduce A→Abc
$aA	de$	shift
$aAd	e$	reduce B→d
$aAB	e$	shift
$aABe	$	reduce S→aABe
$S	$	accept

→ While the primary operations of the parser are shift and reduce there are actually four possible actions a shift-reduce parser can make.

(1) **shift:-**

In shift action, the next input symbol is shifted onto the top of the stack.

(2) **Reduce :-**

In reduce action, the parser knows the right end of the handle is at the top of the stack. It must then locate the left end of the handle within the stack and decide with what non-terminal to replace the handle.

(3) **Accept :-**

The parser announce successful completion of parsing.

(4) Error :-
The parser discover that a syntax error has occured and calls error recovery routine

Very imp.
• Conflict during shift reduce parsing:-

(I) Shift reduce conflict :-
The parser is not able to decide whether to shift or to reduce
For example : A → ab | abcd.
and the parser stack contains $ab and the input buffer contains cd$, the parser cannot decide whether to reduce ab with non-terminal A or shift two more symbols into stack from input buffer.

(II) Reduce-reduce conflict :-
In this case, the parser cannot decide which ~~central~~ sententfal form to use for reduction.
For example : If A → bc and B → abc and the stack contains $abc, the parser cannot decide whether to reduce it into $aA or $B.

• The grammar that leads to shift reduce parser into conflict is known as non-LR grammar. Shift reduce parser can be build successfully for LR grammar and operator grammar.

* Operator precedence parser :-
The operator grammar has the property

that no product on the right side is epsilon (ε) or has no adjacent non-terminal.

Eg: E → EAE | (E) | id
A → + | - | * | /

is not an operator grammar, because the right side EAE has three consecutive non-terminals. However, if we substitute for A then we obtain the following operator grammar.

E → E+E | E-E | E*E | E/E | (E) | id

- Taking an operator grammar we are going to construct operator precedence parser
- The operator precedence parser is the only parser that can parse ambiguous grammar
- 'id' have higher precedence than other operator.
- '$' have lowest precedence than other operator.
- If both operator are same then left side operator have higher precedence because of left ~~recursive~~ associative.

Operation relation table

	id	+	*	$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	<

28th September

* Operator Precedence parser :-

Eg: $E \to E+E \mid E*E \mid id$, Input (w) = id+id*id $

	Input			
	id	+	*	$
id	-	$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	-

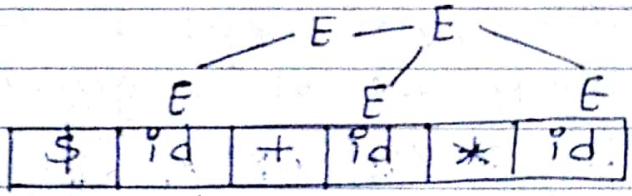
Using this operator relation table we can construct operator precedence parser.

input :- id + id * id $

Bottom up parser

Lookahead

Stack:

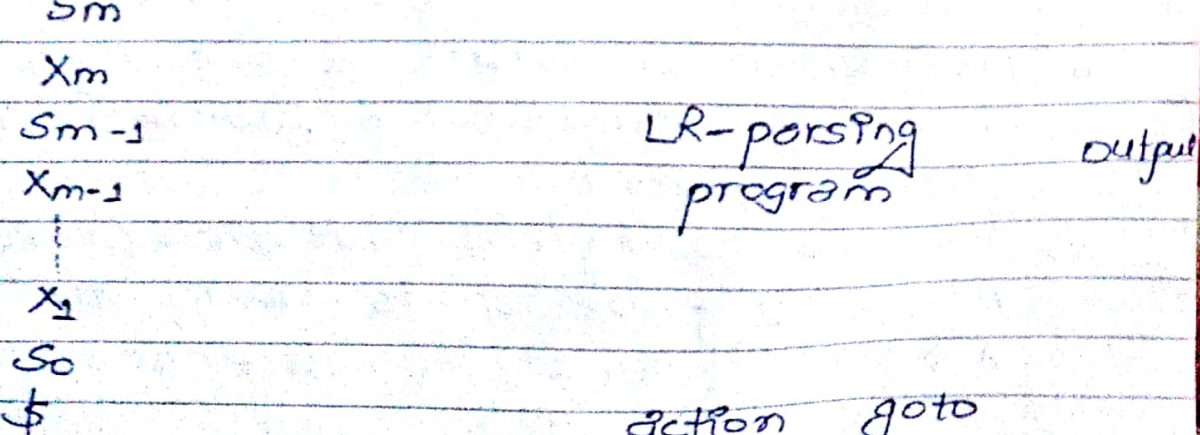


Accepted

- Whenever the top of the stack is less than the lookahead then we have to push it (shift) or when it is greater, than we have to pop it (Reduce).

(Bottom up parser)

* LR parsing :- a_1 a_2 ..... an $



Baki

An LR parser consists of a stack, an input buffer and a parsing table which has two parts (i) goto and (ii) action. Stack consists of the entries of the form $S_0X_1S_1X_2 \cdots X_mS_m$ where S_i is called state and X_i is a grammar symbol either variable or terminal. If top of stack is S_m and input 'a' the LR-parser consists action $[S_m, a]$ which can be one of the four action.

1) shift 'S' where 'S' is a state.
2) Reduce using $A \rightarrow \beta$.
3) Accept.
4) Error.

- The function goto take a state and a grammar symbol as augmented and produces a state.
- The goto function forms the transition table for a DFA.
  - L means that the parser reads input text in one direction typically left to right.
  - R means that the parser produces a reversed rightmost derivation.

- Process to construct LR(0) parsing table:
1) Create the augmented grammar of the given grammar.
2) Compute cannonical collection (LR(0) items)
   $A \rightarrow \cdot XYZ$; parser is ready to scan.
   $A \rightarrow X \cdot YZ$; parser has scanned ex' and

ready to scan 'yz'
$A \to XYZ\cdot$; parser is ready to detect handle.

3) Construct DFA.
4) Construct LR parsing table.

* LR Parser:- (Bottom up parser
  - LR(0) } canonical LR(0) items
  - SLR(1) [simple LR] }
  - LALR(1) [Look ahead LR] } canonical LR(1) items
  - CLR(1) [canonical LR] }

* Construction of LR(0) or SLR(1) parsing table:
To construct the action and goto part for SLR parsing table use the following algorithm:-

1) Given the grammar G, construct an augmented grammar G' by introducing a production $S \to S'$ where 'S' is start variable.
2) Construct closure ($I_0, I_1, \dots I_n$) the collection of sets of LR(0) items for G'.
3) The state "i" is constructed from I_i. The parsing actions for state 'i' are determined as follows.

a) If goto $[I_i, a] = I_j$ then set action $[i, a] =$ "shift j" for terminal 'a'.
b) If $[A \to \alpha\cdot]$ is in I_i then set action $[i, a] =$ "reduce $A \to \alpha$" for all 'a' in follow(A). Here, A should not be S'.
c) If $[S' \to S\cdot]$ is in I_i then set action $[i, \$] =$ accept.

4) If goto $[I_i, A] = I_j$ for non-terminal A, then goto $[i, A] = j$.
5) For all blank entries not defined by rule 3 & 4 set to 'error'.

- Example:

Consider the grammar G.

$S \rightarrow AA$

$A \rightarrow aA / b$

Sol:- The augmented grammar of G is G'

$\therefore G' :- S' \rightarrow S$

$S \rightarrow AA$

$A \rightarrow aA / b$.

Here, computing the LR(0) item set from the above grammar.

i.e. $I_0 = S' \rightarrow \cdot S$

$S \rightarrow \cdot AA$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

Now, taking closure,

I_0 = closure $[S' \rightarrow \cdot S]$

{

$S' \rightarrow \cdot S$

$S \rightarrow \cdot AA$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

Then, goto $(I_0, S) = [S' \rightarrow S \cdot] = I_1$

goto (I_0, A) = closure $(S \rightarrow A \cdot A)$

{

$S \rightarrow A \cdot A$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

} $= I_2$

goto (I_0, a) = closure $(A \rightarrow a \cdot A)$

{

$A \rightarrow a \cdot A$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

$\} = I_3$

goto $(I_0, b) = A \to b\cdot = I_4$

Again,

goto $(I_2, A) = S \to AA\cdot = I_5$

goto (I_2, a) = closure $[A \to a\cdot A]$

$\{$

$A \to a\cdot A$

$A \to \cdot aA$

$A \to \cdot b$

$\} = I_3$

goto $(I_2, b) = A \to b\cdot = I_4$

Again,

goto $(I_3, A) = A \to aA\cdot = I_6$

goto (I_3, a) = closure $(A \to a\cdot A)$

$\{$

$A \to a\cdot A$

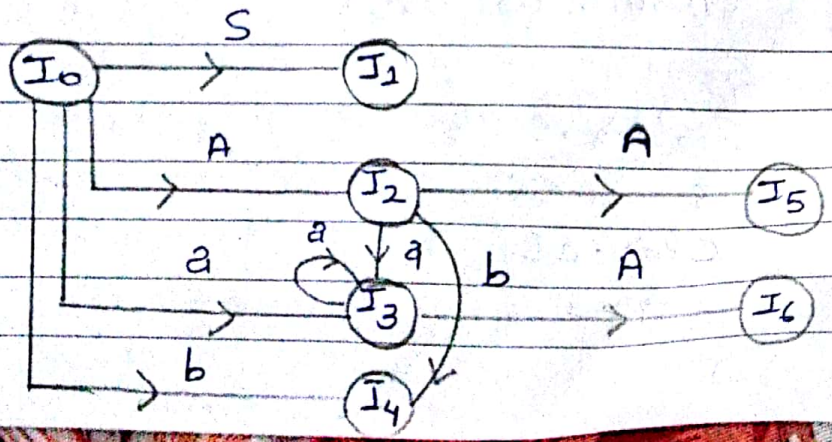
$A \to \cdot aA$

$A \to \cdot b$

$\} = I_3$

goto $(I_3, b) = A \to b\cdot = I_4$

Now, DFA for item set obtained by goto operator is:



9th November

* The closure operation:
If 'I' is a set of items for a grammar 'G' then closure of (I) = set of items constructed from I using following rule:
- If $A \to \alpha \cdot B\beta$ is in closure (I) and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to closure (I) if it is not already there. Apply this rule until no more new items can be added to closure (I).

- The function goto :-
If any item I for all production of the form $A \to \alpha \cdot X\beta$ that are in I.
goto [I, X] is define as the closure of all production of the form $A \to \alpha X \cdot \beta$.

Example :-

$S \to CC$
$C \to cC$
$C \to d$

Ans :- 1) Augumented grammar:

G' :- $S' \to S$
$S \to CC$
$C \to cC$
$C \to d$

Now, the LR(0) items are:

I_0 = closure $(S' \to .S)$

{
$S' \to .S$ (→ S)
$S \to .CC$
$C \to .cC$
$C \to .d$
}

goto (I_0, S) = closure $(S' \to S\cdot) = I_1$

goto (I_0, C) = closure $(S \to C\cdot C)$

$$\{$$
$$S \to C\cdot C$$
$$C \to \cdot CC$$
$$C \to \cdot d$$
$$\}$$
$$= I_2.$$

goto (I_0, c) = closure $(C \to c\cdot C)$ → shifted to right side

$$\{$$
$$C \to c\cdot C$$
$$C \to \cdot cC$$
$$C \to \cdot d$$
$$\}$$
$$= I_3.$$

goto (I_0, d) = closure $(C \to d\cdot) = I_4$

goto (I_2, C) = closure $(S \to CC\cdot) = I_5$

goto (I_2, c) = closure $(C \to c\cdot C)$

$$\{$$
$$C \to c\cdot C$$
$$C \to \cdot cC$$
$$C \to \cdot d$$
$$\}$$
$$= I_3$$

goto (I_2, d) = closure $(C \to d\cdot) = I_4$

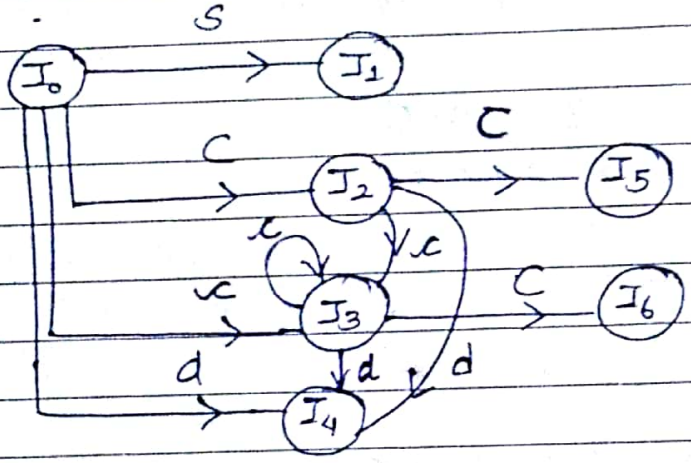
goto (I_3, C) = closure $(C \to cC\cdot) = I_6$

goto (I_3, c) = closure $(C \to c\cdot C)$

$$\{$$

C → c·C
C → ·cC
C → ·d
}
= I_3

goto(I_3, d) = closure(C → d·) = I_4
Nao, transition diagram,



Nao,

- I_0 :-
goto [I_0, C] = I_2 gives goto [0, C] = 2
goto [I_0, c] = I_3 give action [0, c] = shift 3
goto [I_0, d] = I_4 gives action [0, d] = shift 4
goto [I_0, S] = I_1 gives goto [0, S] = 1

- I_2 :-
goto [I_2, C] = I_5 gives goto [2, C] = 5
goto [I_2, c] = I_3 gives action [2, c] = shift 3
goto [I_2, d] = I_4 gives action [2, d] = shift 4

- I_3 :-
goto [I_3, C] = I_6 gives goto [3, C] = 6

goto $[I_3, C] = I_8$ gives action $[3, c]$ = shift 8
goto $[I_3, d] = I_4$ gives action $[3, d]$ = shift 4

- **I_4 :-**
  $C \to d\cdot$
  Follow (C) = {$, c, d}
  action $[4, c]$ = action $[4, d]$ = action $[4, \$]$ = Reduce $C \to d\cdot$

- **I_5 :-**
  $S \to CC\cdot$
  Follow (S) = {$}
  action $[5, \$]$ = Reduce $S \to CC$

- **I_6 :-**
  $C \to cC\cdot$
  Follow (C) = {$, c, d}
  action $[6, \$]$ = action $[6, c]$ = action $[6, d]$ = Reduce $C \to cC$

$S \to CC : r_1$
$C \to cC : r_2$
$C \to d : r_3$

Now, parsing table,

state	action (terminal)			goto (non-terminal)	
	c	d	$	S	C
0	S_3	S_4		1	2
1			Accepted		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

Ex> $E \to E+T/T$
$T \to T*F/F$
$F \to (E)/id$

10th November

* LR(1) Items :-
The general form of an LR(k) items become $(A \to a.b, s)$ where $A \to ab$ is a production $(A \to a.b)$ is called the core and the second part is lookahead. In LR(1) |s| is 1, so 's' is a single terminal.

* Computation of closure :-
For each item of all the form $[A \to \alpha.B\beta, a]$ in I and each production of the form $B \to \gamma$ in G' (Augmented grammar) and each terminal in FIRST(βa) and $B \to .\gamma$ to I if it is not already in the closure(I)

* Computation of goto for LR(1) :-
Given a set of all items of the form $[A \to \alpha.X\beta]$ in I goto $[I, X]$ = closure $([A \to \alpha X.\beta, a)]$

* Computation of LR(1) items :-
1> Start with C = {closure $([S' \to .S, \$])$}
2> Repeat for each items I in C and each grammar symbol 'X' such that goto $[I,X]$ is not already in C, add goto $[I,X]$ to C until no more items set added to C.

(it is compulsory in LR(1))

* Construction of LR(1) parsing table :-

1> Given grammar G, make augmented gramm-ar 'G', introducing $S' \to S$ where S is start variable.

2> Let I_0 = closure ($[S' \to \cdot S, \$]$) starting from I_0, compute the all LR(1) items $I_1, I_2, I_3, \ldots I_n$ and so on.

3> State S_1 is constructed from I_i and the parsing action of state i are defined as.

a> If $[A \to \alpha \cdot a\beta, b]$ is in I_i, and goto $[I_i, a] = I_j$ then set action $[i, a]$ to shift j 'Sj' for each terminal-a.

b> If $[A \to \alpha \cdot, a]$ is in I_i, $A \neq S$ then set action $[i, a]$ to reduce $A \to \alpha$. in lookahead symbol.

c> If $[S' \to S \cdot, \$]$ is in I_i, then set action $[i, \$]$ to accept.

4> The goto transactions for state i are determined as follows:

If goto $[I_i, A] = I_j$ then goto $[i, A] = j$ for each non-terminal 'A'

5> All entries not defined by rule 3 and 4 are made 'error'.

6> The initial state of the parser is the one constructed from the set containing item $[S' \to \cdot S, \$]$

Example :-

$S \to AA$

$A \to aA \mid b$

Sol^n:- Augmented grammar G'

$S' \to S$

$S \to AA$

$A \to aA \quad A \to b$

Now, LR(1) item set are,

I_0 = closure { $[S' \to \cdot S, \$]$

{

$S' \to \cdot S, \$$

$S \to \cdot AA, \$$ (Comparing it with $A \to \alpha \cdot B\beta, a$

$A \to \cdot aA, a/b$ where $A = S, \alpha = \epsilon, B = A, \beta = A$

$A \to \cdot b, a/b$ $a = \$$)

}

goto $[I_0, S] = [S' \to S \cdot, \$] = I_1$

goto $[I_0, A]$ = closure { $[S \to A \cdot A, \$]$ }

{

$S \to A \cdot A, \$$

$A \to \cdot aA, \$$

$A \to \cdot b, \$$

} $= I_2$

goto $[I_0, a]$ = closure { $[A \to a \cdot A, a/b]$ }

{

$A \to a \cdot A, a/b$

$A \to \cdot aA, a/b$

$A \to \cdot b, a/b$.

} $= I_3$.

goto $[I_0, b] = [A \to b \cdot, a/b] = I_4$

goto $[I_2, A] = [S \to AA \cdot, \$] = I_5$

goto $[I_2, a]$ = closure { $[A \to a \cdot A, \$]$ }

{

$A \to a \cdot A, \$$

$A \to \cdot aA, \$$

$A \to .b, \$$

$\mathcal{J} = I_6$

goto $[I_2, b] = [A \to b.\$] = I_7$

goto $[I_3, A] = [A \to aA., a/b] = I_8$

goto $[I_3, a]$ = closure $\{[A \to a.A, a/b]\}$

$\{$

$A \to a.A, a/b$

$A \to .aA, a/b$

$A \to .b$

$\}\ = I_3$

goto $[I_3, b] = [A \to b., a/b] = I_4$

goto $[I_6, A] = [A \to aA., \$] = I_9$

goto $[I_6, a]$ = closure $[A \to a.A, \$]$

$\{$

$A \to a.A, \$$

$A \to .aA, \$$

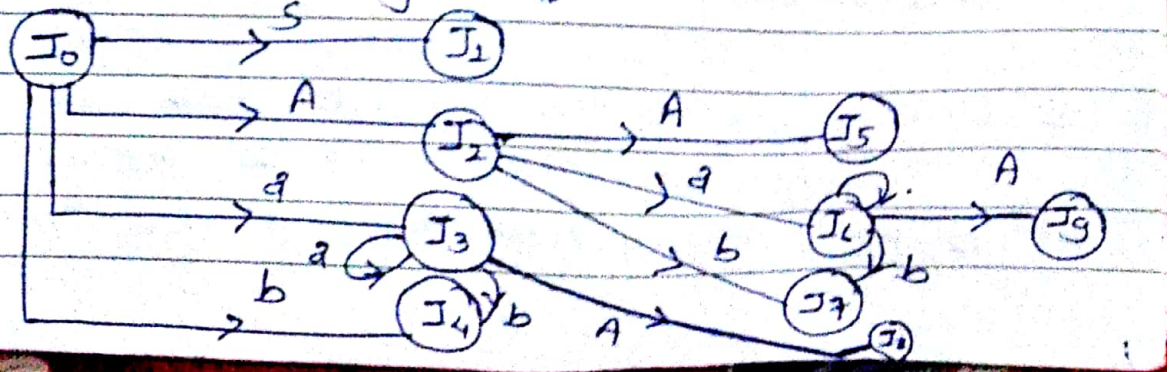
$A \to .b, \$$

$\}\ = I_6$

goto $[I_6, b] = [A \to b., \$] = I_7$

13th November

Transition Diagram :-



* Parsing table :-

	action			goto	
State	a	b	$	A	S
0	S_3	S_4		2	1
1			accepted		
2	S_6	S_7		5	
3	S_3	S_4		8	
4	r_3	r_3			
5			r_1		
6	S_6	S_7		9	
7			r_3		
8	r_2	r_2			
9			r_2		

* LALR : (Lookahead LR) :-

- LALR grammar are mid way in complexity between SLR and LR(1). The table obtained by LALR are considerably smaller than the canonical LR tables.
- The SLR and LALR tables for a grammar always have the same number of states
- They perform a generalization over a LR(1) item sets. A typical programming language generates thousand of states for CLR parser while they generate hundreds of state for SLR and LALR. So, it is much easier to construct a SLR and LALR parser.

* Union operation on Items :-

Given an item set of the form $[A \to \alpha \cdot B\beta, a]$ the first part of the item $A \to \alpha \cdot B\beta$

is called a core of the item. Given two states of the form,

$[I_i = \{A \to \alpha \cdot, a\}]$

$[I_j = \{A \to \alpha \cdot, b\}]$

The core of I_i and I_j is same. The union of two states $I_{ij} = I_i \cup I_j$

$I_{ij} = \{A \to \alpha \cdot, a/b\}$

I_{ij} will perform the reduce operation on seeing either a or b on input buffer. The state machine resulting from the union operation has less states.

* Parsing table for LALR :-

1) Construct $C = \{I_0, I_1, I_2, \ldots, I_n\}$ the collection of sets of LR(1) items.

2) For each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union.

3) Let $C' = \{I_0, I_1, \ldots I_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from I_j in the same manner as in LR(1). If there is a parsing action conflict, the algorithm fails to produce a parser and a gramm-ar is said not to be LALR(1).

4) The goto table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is $J = I_1 \cup I_2 \cup \ldots \cup I_k$, then the cores of goto (I_1, X), goto (I_2, X), ---- goto (I_k, X) are the same, since I_1, I_2

,... I_k all have the same core. Let k be the union of all sets of items having the same core as goto (I_1, X) then goto $(J, X) = k$

- Parsing table:

	action			goto	
State	a	b	$	A	S
0	S_{36}	S_{47}		2	1
1			accepted		
2	S_{36}	S_{47}		5	
3	S_{36}	S_{47}		89	
4	r_3	r_3			
5			r_1		
6	S_{36}	S_{47}		89	
7			r_3		
8	r_2	r_2	r_2		
9			r_2		

In state I_3 and I_6:

$I_3 = A \to a \cdot A, a/b$
$A \to \cdot aA, a/b$
$A \to \cdot b$

$I_6 = A \to a \cdot A, \$$
$A \to \cdot aA, \$$
$A \to \cdot b, \$$

So, $I_3 \cup I_6 = I_{36}$

In state I_4 and I_7:

$I_4 = A \to b \cdot, a/b$ $\quad I_7 = A \to b \cdot, \$$

$\therefore I_4 \cup I_7 = I_{47}$ (core same, look ahead are different so we take it's union)

In state I_8 and I_9:

$I_8 = A \to aA\cdot, a/b$ $I_9 = A \to aA\cdot, \$$

$\therefore I_8 \cup I_9 = I_{89}$.

- Reduce action is only used in lookahead in CLR(1) but it is used in Follow(L.H.S) in SLR / LR(0) so the no. of reduce action is less in CLR(1) parser which reduce the no. of conflict and increase error detecting capacity.

20th November

* Syntax directed translation:-

In any programming language grammar symbols are associated with attributes. The values of these attributes are evaluated by the semantic rules associated with the production rules. The evaluation of these semantics rules may generate immediate code, put information into symbol table, perform type checking, etc. An attribute may be a string, a number, a location or any complex values.

The syntax directed translation indicates the order in which semantic rules are to be evaluated using dependency graph. So, they allow some implementation details to be shown. The conceptual view of syntax directed translation is

input string → parse tree → dependency graph → evaluation order for semantic rule

* Translation schemes :-

The translation scheme indicates the order of evaluation of the semantic actions associated with the production i.e. the translation schemes gives a bit information about implementation details.

A syntax directed definition is a generalization of parse tree where each node in the parse tree has a set of attributes associated with it. The attributes of the parse tree node are either synthesized or inherited. The value of synthesized attribute of a node are determined by the child of that node. The value of inherited attribute of a node is determined by the parent and siblings of the node. The order of these computations depends on the dependency graph induced by the semantic rules. The dependency graph determines the evaluation order of these semantic rules.

Syntax directed definition exchange:

production	semantic rules.
$L \to E/n$	Print (E.val)
$E \to E+T$	E.val = E.val + T.val
$E \to T$	E.val = T.val
$T \to T*F$	T.val = T.val * F.val
$T \to F$	T.val = F.val
$F \to (E)$	F.val = E.val
$F \to digit$	F.val = digit.lexval

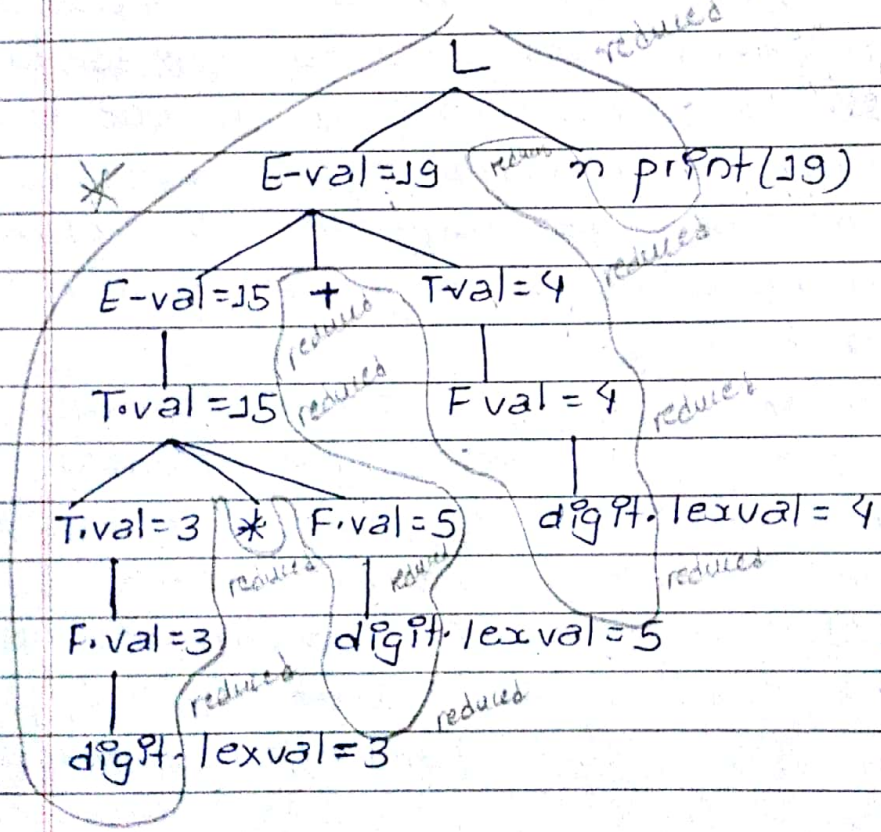
Here, the symbols E, T & F are associated with synthesized attribute value.

VeryImp

* S-attribute definations :-

- An S-attributed defination is a syntax directed defination that uses only synthesized attributes.
- Semantic rules in S-attribute defination can be evaluated by a bottom-up traversal of the parse tree.
- The parse tree for an S-attribute defination can always be annoated by evaluating the semantic rule for attribute for each node in bottom up.

The parse tree with annotation for expression: 3*5+4/n



* Inherited attributes :-

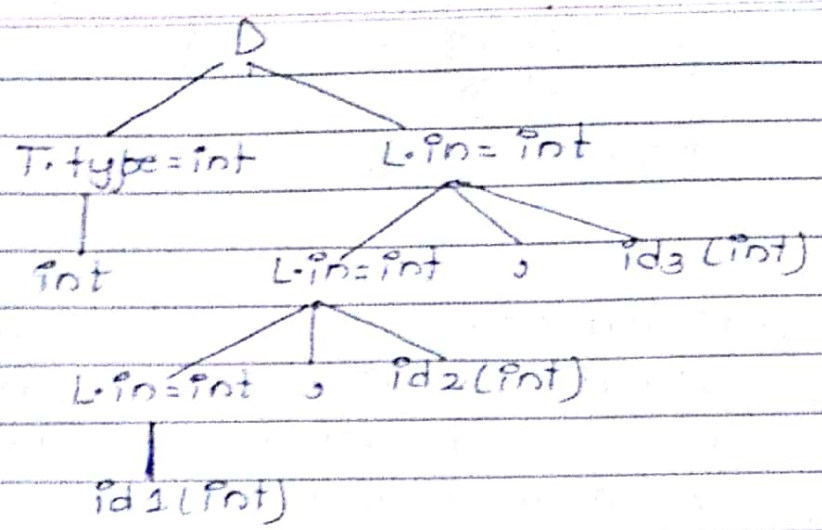
Inherited attribute at any node is defined based on the attributes of parents or siblings of the node. Inherited attributes

are useful for describing context sensitive behaviour of grammar symbol. An inherited attributes can be used to keep track of whether an identifier appears on the left and right side of assignment operator. This can be used to decide whether to use L-value or R-value of the variable.

- Example of inherited attributes:-
  Following example shows the distribution of type information to the various identifiers in the declaration. A declaration generated by a non-terminal D in syntax directed definition consists of keyword 'int' or 'real' followed by list of identifiers 'L'. The non-terminal 'T' has the synthesized attributes type value is determined by the keyword in declaration. The syntax directed definition for this is given below:-

Production	Semantic rules
$D \rightarrow TL$	{ L.type = T.val }
$T \rightarrow int$	{ T.val = int }
$T \rightarrow real$	{ t.val = real }
$L \rightarrow L_1, id$	{ L_1.in = L.in add type [id.entry, L.in] }
~~$L \rightarrow id$~~	
$L \rightarrow id$	{add type (id.entry, L.in)}

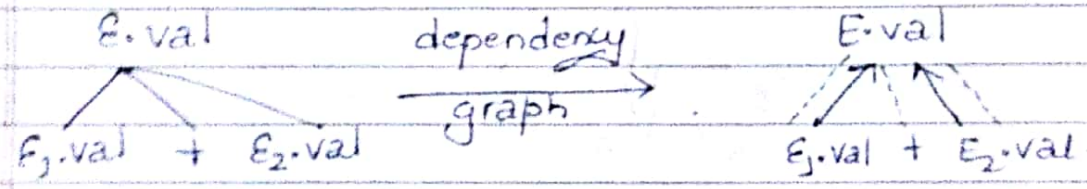
The annotated parse tree for int id_1, id_2, id_3



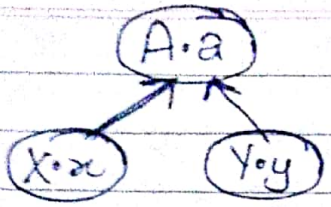
* Dependency graph :-

In order to correctly evaluate the attribute of the syntax tree a dependency graph is useful. A dependency graph is a directed graph that contains attribute as node and dependency across attribute as edges.

$E \rightarrow E_1 + E_2$ { E.val = E_1.val + E_2.val }.



Suppose, A.a = F(X.x, Y.y) is a semantic rule for the production A → xy which defines synthesized attribute X.x, Y.y to which the synthesized attributes A.a depends for the parse tree. For this production there will be three nodes represented by X.x, Y.y and A.a and the attribute of node with A.a depends on attribute X.x and Y.y. So, the dependency graph for this semantic rule will be



24th November

* L-attribute :-
- uses both inherited and synthesized attribute. Each inherited attribute is restricted to inherit either from parent or left sibling only.

Eg: $A \to XYZ \{ X.i = f(A.i), Y.i = f(X.i), Z.i = f(Y.i) \}$

$Y.i = f(Z.i) \}$ → not L-attribute inherit

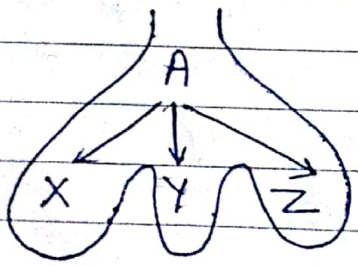
- Semantic rule are placed anywhere on R.H.S.

$A \to \{ \} XYZ$

$| X \{ \} YZ$

$| XYZ \{ \}$

- Attributes are evaluated by traversing a parse tree from depth first, left to right.



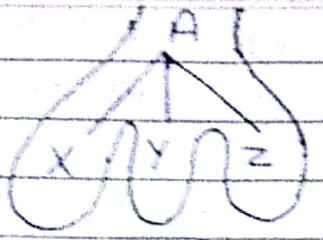
* Difference of S-attribute & with L-attribute :-
- uses only synthesized attribute.

$A \to XYZ \{ A.S = f(X.S), A.S = f(Y.S), A.S = f(Z.S) \}$

- semantic rules are placed at right end of production.

$A \to XYZ \{ \}$

- Attributes are evaluated in bottom up manner.



L-attribute (inherited) → X

$A \to LM \{ L.i = f(A.i), M.i = f(L.s), L.i = f(M.i) \}$

$A \to QR \{ R.i = f(A.s), Q.i = f(R.s), A.s = f(Q.s) \}$

* Bottom up evaluation of s-attribute def^n:

- For bottom-up evaluation of s-attribute defination we use LR parser.
- For bottom-up evaluation of s-attribute defination we put the value of synthesized attribute of the grammar symbol in the stack.

Eg :- Consider the example of SDT:

Production	Semantic rule
$L \to E$	{ L.val = E.val }
$E \to E+T$	{ E.val = E.val + T.val }
$E \to T$	{ E.val = T.val }
$T \to T*F$	{ T.val = T.val * F.val }
$T \to F$	{ T.val = F.val }
$F \to (E)$	{ F.val = E.val }
$F \to$ digit	{ F.val = digit.lexval }

\# For input string 3*5+4 :-

Stack	Input	Attribute	Production
$	3*5+4$	–	shift 3
$3	*5+4$	3	reduce F → digit
$F	*5+4$	3	reduce T → F
$T	*5+4$	3	shift *
$T*	5+4$	3	shift 5
$T*5	+4$	3*5	Reduce F → digit
$T*F	+4$	3*5	Reduce T → T*F

$T	+4$	15	Reduce E→T
$E	+4$	15	Redu shift
$E+	4$	15	shf shift
$E+4	$	15+4	Reduce F→digit
$E+F	$	15+4	Reduce T→F
$E+T	$	15+4	Reduce E→E+T
$E	$	19	Reduce L→E
$L	$	19	—

25th November

* Bottom up evaluation of L-attribute definition:
- While evaluating S-attributed defination in the bottom up parsing the method is straight forward. But the evaluation of inherited attribute is a little bit tricky.

Example:

production	semantic rule
D→TL	{ L.type = T.val }
T→int	{ T.val = integer }
T→float	{ T.val = float }
L→L, id	{settype (id.entry, L.type)}
L→id	{settype (id.entry, L.type)}

Given a string float id, id a bottom up evaluation can be traced as,

stack	input	production
$	float id, id$	shift
$float	id, id$	Reduce T→float
$T{T.val= float}	id, id$	shift
$Tid	, id$	Reduce L→id
$TL{L.type= T.val}	, id$	shift
$TL,	id$	shift
$TL, id	$	Reduce L→L, id

Stack	input	Production.
$TL{L.type = T.val}	$	reduce D → TL
$D	$	—

strings accepted.

* **Type checking :-**

Token → Parser —syntax tree→ type checker —syntax tree→ intermediate code generator ——→ intermediate representation.

- A compiler has to do semantic checks in addition to syntax analysis.
- A type system is a collection of rules for assigning type expression to the parts of program.
- The type checker implements the type system

- **Type expression :-**

The type of a language construct is defined by a type expression. A type expression is defined as follows:

- A basic type is a type expression for example integer, character, real, etc.
- A type name is a type expression example if int is named by a variable x then x is a type expression.
- A type construct applied to a type expression is a type expression.

Example: T array [I] or array [I,J] is the type expression with I element of type T, int a[10]

- If T_1 and T_2 are type expressions then cartesian product $T_1 \times T_2$ is also a type expression

* A simple language :- (Grammar for source language)

$P \rightarrow D ; E$
$D \rightarrow D ; D \mid id , T$
$T \rightarrow char \mid integer \mid array [num] of T \mid * T$
$E \rightarrow Literal \mid num \mid id \mid E_1 op E_2 \mid * E$

The above grammar generates programs represented by the non-terminal P, consisting of a sequence of declaration D, followed by a single expression E.

Specification of simple type checker :-

The translation schemes that saves the type of an identifier.

Production	Semantic rule
$P \rightarrow D ; E$	
$D \rightarrow D ; D$	
$D \rightarrow id , T$	{ addtype (id.entry, T.type)}
$T \rightarrow char$	{ T.type = char }
$T \rightarrow integer$	{ T.type = Int }
$T \rightarrow * T$	{ T.type = pointer (T_1.type)}
$T \rightarrow array [num] of T_1$	{ T.type = array (1...num.val, T_1.type) }

In the translation scheme, the action associated with the production $D \rightarrow id , T$ saves a type in a symbol table entry for an identifier.

The action addtype (id.entry, T.type) is applied to synthesized attribute entry pointing to the symbol table entry for id and

a type expression by synthesized attribute type of non-terminal T.

27th November

* Type checking of expression :-

$E \rightarrow id$	$\{E.type = id.entry\}$
$E \rightarrow literal$	$\{E.type = char\}$
$E \rightarrow num$	$\{E.type = integer\}$
$E \rightarrow E_1 OP E_2$	$\{E.type =$ if $E_1.type =$ integer and $E_2.type =$ integer then integer else type error$\}$
$E \rightarrow *E_1$	$\{E.type =$ if $E_1.type =$ pointer(t) then t else type error$\}$

In the above example, the type checking for the expression is described by the semantic rule associated with each production. Consider a following grammar for arithmetic expression using an operator OP to integer or real numbers.

$E \rightarrow E_1 OP E_2 \mid num \mid id$

Give the syntax directed defination to determine the type expression. when two integers are fused in the resulting type is integer otherwise it is real.

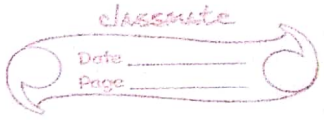
Given SDD,

$E \rightarrow id$ $\{E.type = id.entry\}$

$E \rightarrow num$ $\{E.type = int\}$

$E \rightarrow E_1 OP E_2$ $\{E.type =$ if ($E_1.type = int$ & $E_2.type = int$) then int

else if ($E_1.type = int$ and $E_2.type = real$) then real

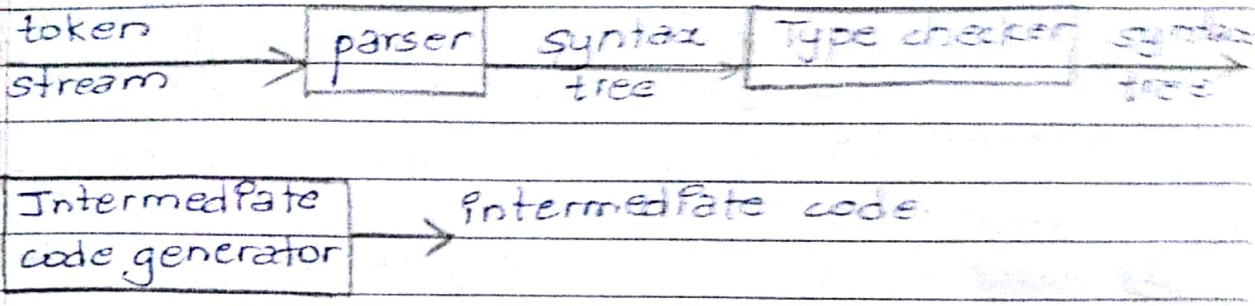



```
else if ( E1.type = real and E2.type = real ) then real;
else if ( E1.type = real and E2.type = int ) then real
else
    type error();
```

UNIT-3

Intermediate Code Generator

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation from which the back end generates target code.



* **Three address code :-**

A three address code is a sequence of statement of the general form $x = y \ op \ z$ where x, y & z are names constants for a compiler generated temporaries and op for an operator. So, the source language expression like $x+y*z$ might be translated into sequence as:

$t_1 = y * z$, $t_2 = x + t_1$ where t_1 & t_2 are compiler generated temporaries.

Similarly, for $x = y + z * w$ should be represented as:

$t_1 = z * w$, $t_2 = y + t_1$, $x = t_2$

* **Types of 3-address code statements :-**

(1) **Assignment statement :-**

$x = y \ op \ z$ where op is binary operator y and z are operands. The result of y and z is stored in x.

② Unary operation :-
X = OPZ. This is used for unary minus.
Eg: X = Y * (-Z) + K
Three address code for the above example will be,

$t_1 = -Z$
$t_2 = Y * t_1$
$t_3 = t_2 + K$
$X = t_3$

③ Copy statement :- a=b [The value of b is stored in variable a]

④ Unconditional jump :- goto 4 [Here 4 is a label]

⑤ Conditional jump :- if x relop y goto 1 (relop = relational operator)

⑥ Function call :- For a function 'fun' with n argument $a_1, a_2, \dots, a_n$
fun $(a_1, a_2, \dots a_n)$ the three address code will be

param a_1
param a_2
. . . . . .
. . . . . .
param a_n
call fun, n [∵ where param defines the argument to function]

Eg:
```
int fun (int a, int b)
{
  return a+b;
}
```

```
int main()
{
  int c;
  int d;
  fun(c,d)
}
```

Using three address code

```
fun:
t0 = a+b
Return t0
End fun
Main
  param c;
  param d;
  t1 = Lcall - fun.
```

* Array Index :-

In order to access the elements of array either single or multi-dimensional, three address code require base address and offset value.

Eg: X = Y[i]

Memory location M = Base address of Y + displacement i

X = content of Memory location M.

X[j] = Y

Memory location M = Base address of X + displacement j.

The value of y is stored in memory location M.

* Pointer assignment :-

X = &y X stores the address of memory location y.

X = *y y is a pointer whose r-value is at location X.

*X = y r-value of the object pointed by X to the r-value-Y.

2nd December

* Implementation of 3-address code :-
A three address is an abstract form of intermediate code. In a compiler, these statements can be implemented as record with field for the operator and operand.

Following are the three code representation :-

(1.) Quadruple :-
- A quadruple is a record structure with four fields which we call op, arg1, arg2, and X in result.
- The three address statement X = Y op Z is represented by placing Y in arg1, and Z in arg2, and X in result.
- Statement with unary operator like X = -Y or X = Y do not use arg 2
- operator like param use neither arg2 nor result.
- conditional and unconditional jump put the target label in result.
- Eg :- statement a = b * -c + b * -c

X = Y op Z.

	op	arg 1	arg 2	result
0	u. minus	c		t_1
1	*	b	t1	t_2
2	u. minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	:=	t_5		a

(2.) Triples :-
To avoid entering temporary names into the symbol table we might refer to a temporary

value by the position of the statement that computes it.

Example:

	OP	arg 1	arg 2
(0)	u.minus	c	
(1)	*	b	(0)
(2)	u.minus	c	(2)
(3)	*	b	(3)
(4)	+	(1)	(3)
(5)	:=	a	(4)

3. Indirect triples :-

It considers listing pointer to triples rather than listing triples themselves. This implementation is mutually called triples.

	Statement	op	arg 1	arg 2
(0)	(10)	(10)	u.minus	(a)
(1)	(11)	(11)	*	(b)
(2)	(12)	(12)	u.minus	(c)
(3)	(13)	(13)	*	(d)
(4)	(14)	(14)	+	(e)
(5)	(15)	(15)	:=	(f)

* Syntax directed translation into 3-address code :-

To represent the syntax directed translation into 3-address, the semantic rules for translation can be written as follows :-

- Firstly deal with assignments.
- Use the attributes.
  - E.place = the name that will hold the value of E.

• E.code = holds the 3-address code statements that evaluate E.

- use function new-temp that returns a new temporary variable that we can use.
- use function go-to generate the single 3-address statement given the necessary information.
- For the moment, we create a new name every-time a temporary is needed.

* Boolean expression :-

They are use to either complete logical values or as conditional expression in flow of control statement.

We consider boolean expression with the following grammar:

$E \rightarrow E$ or $E \mid E$

and, $E \mid$ not $E \mid (E) \mid$ id relop id $\mid$ true $\mid$

There are two methods to evaluate boolean expression. They are :-

1) Numerical representation :-

In this method, implementation of boolean expression using '1' to denote true and '0' to denote false.

2) Jumping code :-

We represent the value of boolean expression by the position reached in the program.

Eg: The translation for a or b and not c is

t_1 = not c

t_2 = b and t_1

t_3 = a or t_2

A relation expression such as a < b is equivalent to conditional statement if a < b then 1 else 0. It's translation into 3-address code can be represented by jumping code:

100 :- if a<b goto 103 (JMP 103)
101 :- t = 0
102 :- goto 104.
103 :- t = 1.

translation

12th December continue

A transition scheme for boolean expression producing three-address code for boolean expression:

In this scheme, we assume that emit places three address statement into an output file in the right format. The next state gives the index of the next three address statement in the output sequence.

$E \to E_1$ or E_2 { E.place = new temp, emit (E.place = E_1.place 'or' E_2.place) }

$E \to E_1$ and E_2 { E.place = newtemp, emit (E.place = E_1.place and E_2.place) }

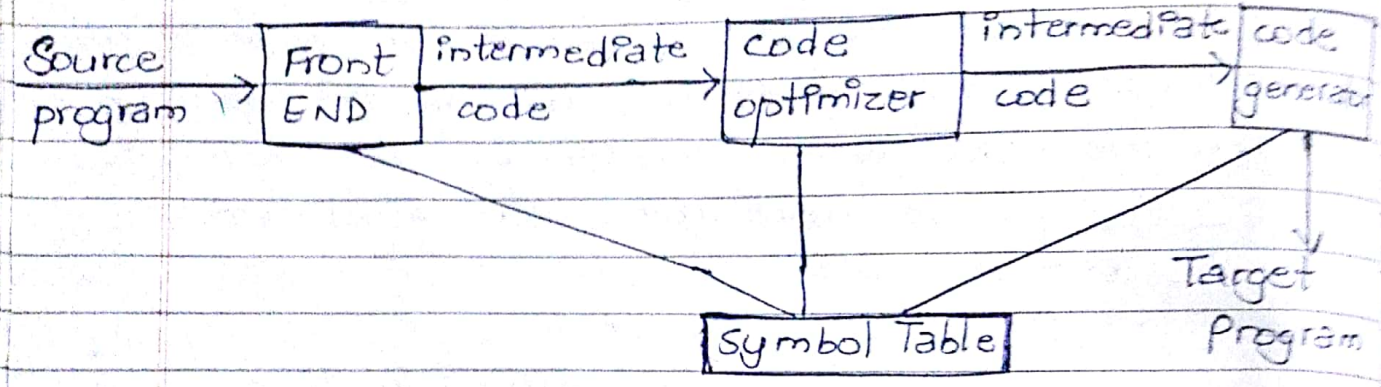
$E \to$ not E_1 { E.place = newtemp emit (E.place = not E_1.place) }

$E \to id_1$ relop id_2 { E.place = new temp,
emit (if id1.place relop id2.place
goto nextstate + 3)
emit (E.place = 0)
emit (gotonext + 2)
emit (E.place = 1) }

E → true { E.place = newtemp
emit (E.place = 1) }

E → false { E.place = newtemp
emit (E.place = 0) }

CODE GENERATOR



- The final phase in compiler model is the code generator.
- It takes input as an intermediate representation of the source program and produces as output equivalent target program.

V.V. Imp
T.U. 2069
T.U. 2072

* Code generator design issue :-
The details of code generation are dependent on the target language and operating system issues such as memory management, instruction selection, register allocation, evaluation order are almost all code generation problems.

① Input to code generator:-

The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with information in symbol table. For example:- intermediate representation with the information into symbol table as postfix, 3-address code etc.

② Target Program:-

- The output of the code generator is the target program.
- The output of code generator i.e. target program may take variety of forms such as absolute machine language, relocatable machine code.
- Producing an absolute machine language as output has the advantage that it can be placed in a fixed location in memory and immediately execute. A small program can be compiled and executed ~~easily~~. quickly.
- Producing a relocatable machine language as output arrows subprogram to be compiled separately.

③ Intruction Selection:-

Efficient and low cost instruction selection is important for code generation.

Example: Three address statement of the form, $X := Y + Z$ where a, y and z are statistically allocated and can be translated into the code sequence.

```
MOV y, R0      /* Load Y into register R0 */
ADD z, R0      /* Add z to R0 */
MOV R0, X      /* Store R0 into X */
```

Unfortunately, statement-by-statement code generation often produce poor code.

For example: a := b+c.

d := a+e.

```
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0
ADD e, R0
MOV R0, d
```

Here, third and fourth statements are redundant.

So,
```
MOV b, R0
ADD c, R0
ADD e, R0
MOV R0, d.
```

(4) Register allocation:-

- Instructions involving register operands are usually shorter and faster than those involving operands in memory.
- The proper utilization of register certainly increase the efficiency of the program.

Eg: t := a+b
t := t+c
t := t/d.

Here,
```
L R1, a
A R1, b
M R0, c
A R1, R0.
```

D R1, d
ST t, R1

L R0, a
A R0, b
A R0, c
D R0, d
ST t, R0

(5) Order of evaluation :-
The order of evaluation of instruction for any expression is important for correctness and efficiency.
• The order in which the computations are performed can affect the efficiency of the target code.

* Target Machine :-
Our target computer is a byte addressable machine with four byte to a word and n general purpose register $R_0, R_1, \cdots R_{n-1}$. It has two address instruction of the form:-
OP source, destination.
in which OP is an op-code and source and destination are data fields.
MOV : move source to destination.
ADD : add source to destination.
SUB : subtract source from destination.
MUL : multiply source ~~and~~ with destination.
The source and destination of instructions are specified by combining register and memory location with address mode.

16th December

* Target Machine:-

The address mode together with their assembly code, form and associated cost are:

Mode	form	Address	cost
Absolute	M	M	1
Register	R	R	0
Indexed	c(R)	c+contents(R)	1
Ind.Register	*R	content(R)	0
Ind.Indexed	*c(R)	Contents (c+ contents (R))	1
Literal	#c	c	1

- A memory location M or a register R represents itself when used as a source or destination.

  Eg: MOV R_0, M /* stores the contents of Register R_0 into memory location M */

- An address offset c from the value in Register R is written as c(R)

  Eg: MOV 4(R0), M /* store the values (4 + contents(R0) in memory location M */

- Indirect version of modes are indicated by prefix '*'

  Eg: MOV *4(R0), M /* store the values/contents (4+contents (R0)) into memory location M */

- Literal:

  Eg: MOV #1, R0 /* Load the constant 1 into register R0 */

* Instruction cost:-

We take the cost of an instruction to be

one plus the cost associated with the source and destination address mode.

- Address mode involving register have cost zero, while those with memory location or literal in them have cost one.

Example:

```
MOV  R0, R1   → cost 1
MOV  R0, M    → cost 2
MOV  4(R0), M → cost 3
```

* Instruction Selection :-

Let, we have 3-address code like x=y+2. The target machine code for their 3-address statement instructions.

		Cost
MOV	(y), R0 (y → literals)	1+1+0 = 2
ADD	z, R0	1+1+0 = 2
MOV	R0, x	1+0+1 = 2
		Total = 6.

Assuming x, y, z are contents of register R0, R1, R2 then,

	Cost
ADD *R1, *R2	1+0+0 = 1
MOV *R2, *R0	1+0+0 = 1
	Total = 2.

We can generate good code for the machine by utilizing its addressing capabilities efficiently.

* Run time storage management :-

Run time storage management requireds following entities:

a) Code :-

It is known as the text part of the program that does not change at runtime. It's memory requirement are known at the compile time.

b) Procedures :-

Their text part is static but they are called in a random manner. That is why, stack ~~m~~ storage is used to manage procedure calls and activations.

c) Variables :-

Are known at the runtime only. Heap allocation is used for managing allocation and de-allocation of memory for variables in runtime.

A compiler gets a block of memory from the OS for the compiled program to run.

a) Static allocation :-

fixed or static data

In this allocation scheme, the compilation data is bound to fixed location in the memory and it does not change when the program executes. As the memory requirements and storage location are known in advance, runtime supports package for memory allocation and de-allocation is not required.

b) Stack Allocation :-

dynamic data

Procedure calls and their activation are managed by means of stack memory allocation. It works in LIFO and is very useful for recursive procedure calls.

c) Heap ~~collection~~ Allocation:-

Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variable are no more required.

* Basic Blocks and Flow Graphs:-

A basic block is a sequence of consecutive statement in which flow of control enters at the beginning and leaves at the end without halting or branching except at the end.

Example:

$a = b+c+d$

The three-address code is,

$t_1 = b+c$

$t_2 = t_1+d$

$a = t_2$.

The basic blocks does not have any jump statements among them.

Example:

If A<B then 1, else 0

I) If A<B goto (4)

II) $T_1 = 0$

III) goto (5)

IV) $T_1 = 1$

V) .....

After an intermediate code is generated for the given code, we can use the following rules to partition into basic blocks:

Rule:1 Determine the leader:

a) The first statement is a leader.

b) Any target statement of conditional or uncon-

nditional goto is a leader.
c) Any statement that immediately follows a goto is a leader.

Rule:2 The basic block is formed starting at the leader statement and ending just before the next leader statement appearing.

Example:

1) PROD = 0	B_1
2) I = 1	
3) T_2 = addr(A) - 4	
4) T_4 = addr(B) - 4	

leader

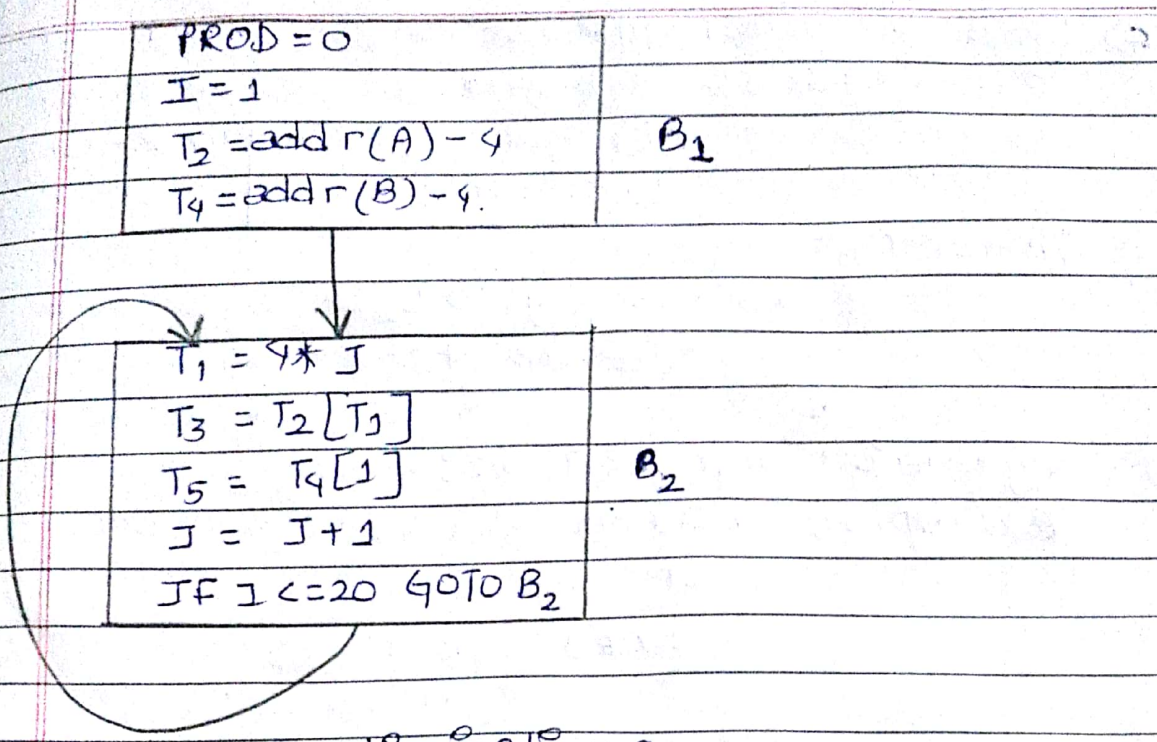
5) T_1 = 4 * I	B_2
6) T_3 = T_2[T_1]	
7) T_5 = T_4[T_1]	
8) I = I + 1	
9) IF I <= 20 Goto (5)	

leader

* Flow graph :-
A flow graph is a directed graph in which the flow control information is added to the basic blocks.

- Rules :- The basic blocks are the nodes to the flow graphs.
  - The block whose leader is the first statement is called initial block.
  - There is a directed edge from block B_1 to Block B_2 if B_2 immediately follows B_1 in the given sequence, we can say that B_1 is a predecessor of B_2

Now, the flow graph for the three address code:



* Peephole optimization :-
- A simple but effective technique for locally improving the target code is peephole optimization.
- A method for trying to improve the performance of the target program by examining a sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence.

- Some peephole optimization technique :-

a) Elimination of redundant instruction :-

Eg: LDA R0, A
ST A, R0.

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading & storing may carry the same meaning even if some of them are removed.

b) Elimination of unreachable :-
↳ part of program code that is never accessed bcz of programming construct.

Eg: if a<b goto L1
goto L2 } unreachable
a = b
L1 : a = b + 2
L2 : b = a + 2

#Back Patching
code generator generates.
Generating code from dags.

c) Flow of control optimization :-

There are instances in a code where the program control jumps back and forth without performing any tasks. These jumps can be removed.

Example : If a<b goto L_1
L_1 : goto L_2 } → unnecessary jump
L_2 : a = b+2

Therefore,
If a<b goto L_2
L_2 : a = b+2.

d) Algebraic Simplification :-
Example : x = x+0 can be replaced by x itself
or
x = x*1 can be replaced by INC x

e) Use of machine Idioms :-
It reduces the execution time and improve the code.

Eg : i = i+1
INC i
i = i-1
DEC i

Unit: 3.3

* Code optimization:-

- The principal source of optimization:
  - A transformation of a program is called local if it can be performed by looking only at the statement in a basic block otherwise it is called global.
  - Many transformation can be performed at both the local and global levels. Local transformation are usually performed first.

- Code optimization technique:-

(I) Dead code elimination.
(II) Constant propagation and constant folding.
(III) Common sub-expression elimination.
(IV) Code motion.

(I) Dead code elimination:-

Eliminates code that cannot be reached or where the result are not subsequently used.

for example: int count

```
void A
{
 int i;
 i=1; // deadcode since it is
        not subsequently used //
count=1; // deadcode since it was
           overwritten //
count=2;
return;
count=3; // deadcode since
           unreachable //
}
```

(II) Constant propagation and constant folding :-
Constant folding refers to the technique of evaluating constant operands at compile time. Eg: int f(void) { return 3+5; } → return 8;

- Constant propagation :- After constant folding → return 8;
If a variable is assigned a constant value then subsequent use of that variable can be replaced by a constant as long as no intervening assignment has changed the value of the variable.

For example : int x = 12;
int y = 7 - x/2;
return y * (24/x + 2)

Applying constant propagation :- int x = 12;
int y = 7 - 12/2;
return y * (24/12 + 2)

Applying constant folding we have
int x = 12;
int y = 1;
return 1 * 4 = 4

(III) Common ~~sub-elimination~~ sub-Expression Elimination :-
This is a code optimization technique that scans the code to find identical expressions and replaces redundant expression each time it is encountered.

For example : a = b*c + g;
d = b*c + e;

Here, we see that the sub-expression b*c repeats so we can transform the code to

```
t = b*c;
a = t+g;
d = t+e;
```

(iv) Code motion :-

It is also called loop-invariant. Code motion has to do with moving a block of code outside a loop if it won't have any difference if it is executed outside or inside the loop.

Example :
```
for (int i = 0; i<n; i++)
{
    x = y+z;
    a[i] = 6*i;
}
```

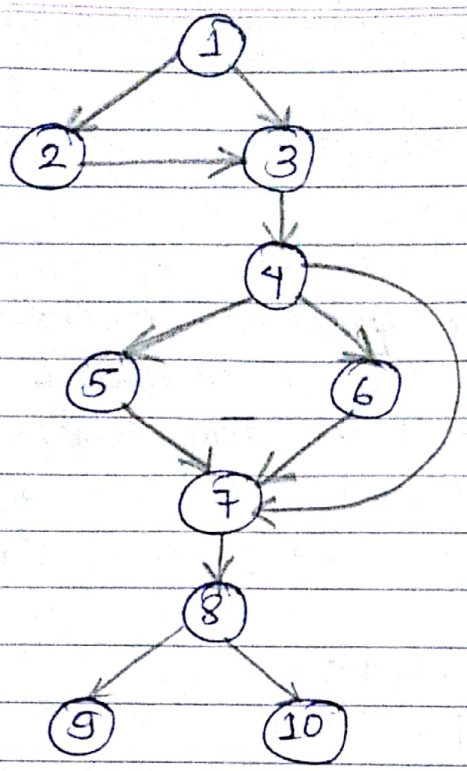
In the code fragment, the expression x = y+z has no effect inside the loop and can safely be moved outside of the loop, the resulting code would be,

```
x = y+z;
for (int i=0; i<n; i++)
{
    a[i] = 6*i;
}
```

- ✳ Loops in Flow Graph :-
- Dominator :-

Node 'd' of a flow graph dominates node 'n' which can be written as d domn, if every path from the initial node of the flow graph to n goes through 'd'. Every node dominates itself and entry of the loop dominates all nodes in the loop.

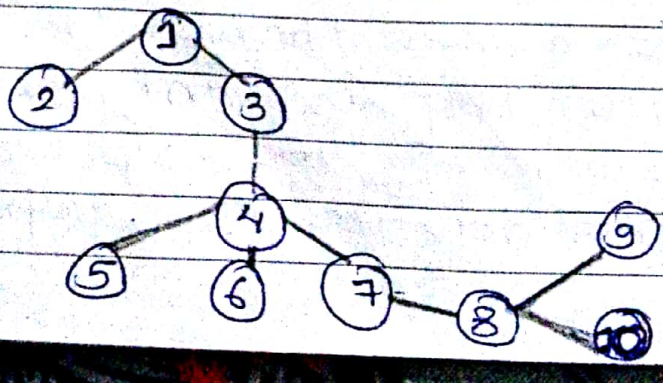
Example : Consider the flow graph,



- Initial node (1) dominates every node.
- Node (2) dominates only itself because control can reach to node (3) without node (2).
- Node (3) dominates all but not node (1) and (2).
- Node (4) dominates all but not node (1), (2) and (3).
- Node (5) and (6) only dominate themselves, since flow of control can slap around either by going through the other.

Finally, node (7) dominates 7, 8, 9 and 10, and node (8) dominates (9) and (10).

Now, dominator tree for flow graph.



Assignment:

1) Generating code from DAGS:-

- A DAG is a ~~basic~~ directed acyclic graph with the following labels or nodes:

a) Leaves are labeled by unique identifiers, ~~for labels to store~~ the either variable names or constants.

b) Interior nodes are ~~the~~ labeled by an operator symbol.

c) Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

- DAGS are useful data structures for implementing
- transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsiquent statements.
- It provides a good way of determining common sub-expression.
- The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than ~~we can~~ starting from a linear sequence of three-address statements or quadruples.

② Rearranging the order:-

The order in which computations are done can affect the cost of resulting object code. for example, consider the following basic block:

$t_1 := a+b$

$t_2 := c+d$

$t_3 := e - t_2$
$t_4 := t_1 - t_3$

Generated code sequence for basic block:

```
MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
MOV R0, t1
MOV e, R0
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
```

Rearranged basic block:
Now t_1 occurs immediately before t_4.

$t_2 := c + d$
$t_3 := e - t_2$
$t_1 := a + b$
$t_4 := t_1 - t_3$

Revised code sequence:

```
MOV c, R0
ADD d, R0
MOV a, R0
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R0, t4
```

In this order, two instruction MOV R_0, t_1 and MOV t_1, R_1 have been saved.

* Backpatching :-

- It is the technique to solve the problem of symbolic names in goto statements by the actual target addresses. This problem comes up because of, some language do not allow symbolic name in the branches.

Example:

Source:

```
if a or b then
  if c then
  x = y+1
```

Translation:

```
    if a goto L1
    if b goto L1
       goto L3
L1 : if c goto L2
       goto L3
L2 : x = y+1
L3 :
```

After Backpatching

```
100 : if a goto 103
101 : if b goto 103
      goto 106.
```

20th December

* LR(1)

Given grammar:

$S \to CC$

$C \to cC$

$C \to d$

Augumented grammar:

$S' \to S$

$S \to CC$

$C \to cC$

$C \to d$

Now, I_0 = closure $\{S' \to \cdot S, \$\}$

{

$S' \to \cdot S, \$$

$S \to \cdot CC, \$$

$C \to \cdot cC, c/d$

$C \to \cdot d, c/d$

}

goto $[I_0, S] = [S' \to S \cdot, \$] = I_1$

goto $[I_0, C]$ = closure $\{S \to C \cdot C, \$\}$

{

$S \to C \cdot C, \$$

$C \to \cdot cC, \$$

$C \to \cdot d, \$$

$\} = I_2$

goto $[I_0, c]$ = closure $\{C \to c \cdot C, c/d\}$

{

$C \to c \cdot C, c/d$

$C \to \cdot cC, c/d$

$C \to \cdot d, c/d$

$\} = I_3$

goto[I_6, d] = C → d·, $ = I_7

Now, the parsing table is:

State	Action			Goto	
	c	d	$	S	C
0	S_3	S_4		1	2
1			Accepted		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5					
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

T.U. 2068

3. Convert the regular expression 0+(1+0)*00 first into NFA and then into DFA using Thomson's and subset Construction Methods.

Sol:-

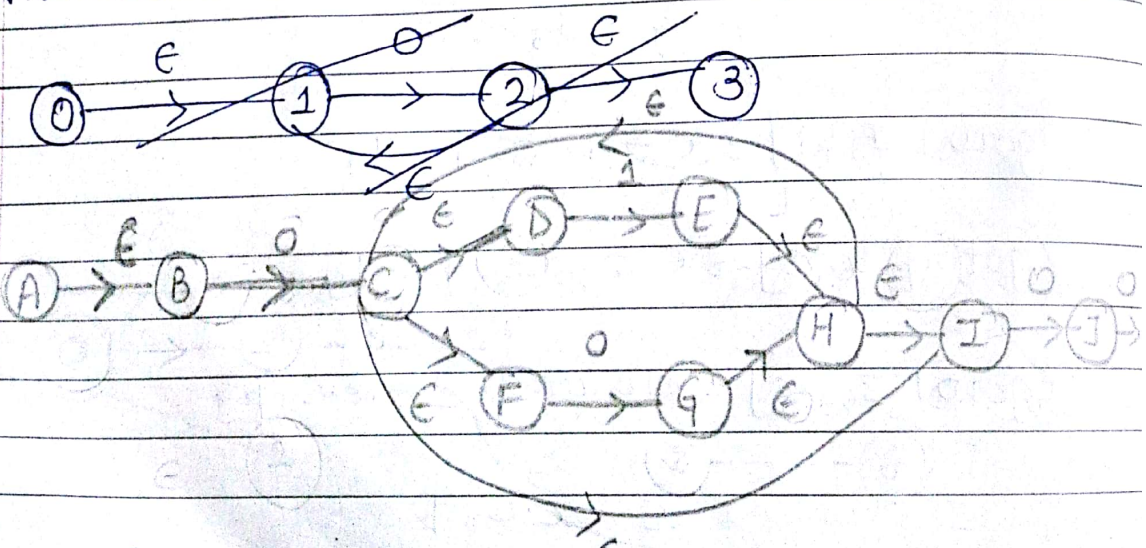


Fig:- Construction of NFA.

Now, Constructing the DFA from NFA using Thomson subset Construction:

Date 13

$\epsilon \rightarrow$ See Follow
Non-$\epsilon \rightarrow$ See First

FIRST and FOLLOW:

* Calculate FIRST:
- FIRST(T) = {T}
- FIRST(E) = {E}
- FIRST(N) = FIRST(N)

Example:

$E \rightarrow TE'$
$E' \rightarrow TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id$.

Soln:
FIRST(E) = { (, id }
" (E') = { (, id, ϵ }
" (T') = { *, ϵ }
" (F) = { (, id }
" (TE') = { (, id }
" (E) = { ϵ }
" (*FT') = { * }
" ((E)) = { (}
" (id) = { id }

Follow:
(i) If $S \rightarrow \alpha$ then follow (S) = $
(ii) If $A \rightarrow \alpha B \beta$ then Follow (B) = FIRST(β) except ϵ
(iii) If $A \rightarrow \alpha B$ then Follow (B) = Follow (A)

Now,
Follow (E) = { $,) }
Follow (E') = { $,) }
" (T) = { (, id, $,) }
" (T') = { (, id, $,) }
" (F) = { *, (, id, $,) }

I_1

- goto $(I_0, E) = F' \to E \cdot$
  $E \to E \cdot + T$

- goto $(I_0, T) = E \to T \cdot$
  $= T \to T \cdot * F$
  $= I_2$

- goto $(I_0, F) = T \to F \cdot$
  $= I_3$

- goto $(I_0, ()) = F \to (\cdot E)$
  $= E \to \cdot E + T$
  $= E \to \cdot T$
  $= T \to \cdot T * F$
  $= F \to \cdot (E)$
  $= F \to \cdot id$
  $= I_4$

- goto $(I_0, id) = F \to id \cdot$
  $= I_5$

- goto $(I_1, +) = E \to E + \cdot T$
  $T \to \cdot T * F$
  $T \to \cdot F$
  $F \to \cdot (E)$
  $F \to \cdot id$
  $= I_6$

- goto $(I_2, *) = T \to T * \cdot F$
  $F \to \cdot (E)$
  $F \to \cdot id$
  $= I_7$

- goto $(I_6, E) = F \to (E \cdot)$
  $= E \to E \cdot + T$
  $= I_8$

- goto $(I_6, T) = E \to E + T \cdot$
  $T \to T \cdot * F$
  $= I_9$

- goto $(I_7, F) = T \to T * F \cdot$
  $= I_{10}$

- goto $(I_8,)) = F \to (E) \cdot$
  $= I_{11}$

- goto $(I_8, +)$

* Lookahead rule for LR(1) collection:
  $A \to \alpha \cdot B\beta, a$
  FIRST (βa)

* Major Parts of compiler:-
Two major parts:-
a) Analysis Part.
b) Synthesize Part.
- In analysis phase, an intermediate representation is created from the given source program.
- Lexical Analyzer, Syntax Analyzer and the semantic analyzer are the parts of analysis phase.

- In the synthesis phase, the equivalent target program is created from the intermediate representation.

- Intermediate code generator, code generator and the code optimizer are the parts of synthesis phase.

IU-2068

* Analysis of source program :-

V.V.Imp

1) Lexical Analysis :-

- The stream of characters forming the source program are scanned linearly to produce the stream of logical element called tokens.
- i.e. lexical analyzer reads the source program and returns the tokens of the source program.
- A token describes the patterns of characters having same meaning in the source program.
- Puts information about the identifiers into the symbol table.
- Regular expressions are used to describe tokens.

Example: newval := oldval + 12

Here,

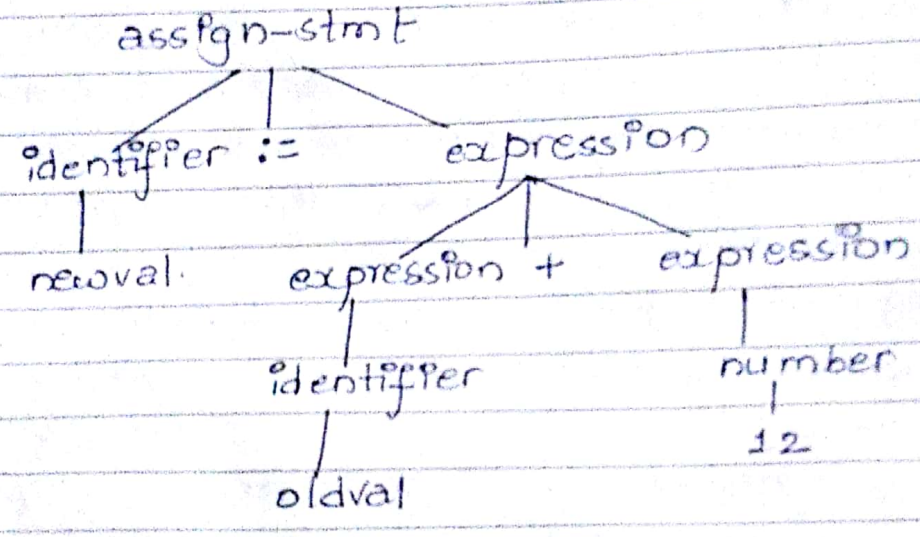
newval	is	identifier.
:=	is	assignment operator.
oldval	is	identifier.
+	is	add operator.
12	is	number.

2) Syntax Analyzer :-

- creates the syntatic structure of the given program
- Also called a parser.
- i.e. in syntax analysis, the stream of tokens are grouped in a hierarchical collection forming the syntax grouping.
- A parse tree describes a syntatic structure.
- The syntax of a language is specified by a CFG.

- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether the given program satisfies the rules implied by a CFG.
- If it satisfies the rule then a parse tree is created for the given program.

Ex:- assign-stmt := expression
expression → identifier
expression → number
expression → expression + expression



3) Semantic Analyser:-

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type checking is an important part of the semantic analyzer.
- Normally, semantic information cannot be represented by the context free language used in the syntax analyzers.
- The result of a semantic analyzer is a syntax directed translation.

Example:- newval := oldval + 12

Here, the type of identifier - newval must have or match with the type of expression (oldval + 12)

Synthesis of source Program :-

- Consists of following phases :
1) Intermediate code Generation.
2) Code optimization.
3) Code Generator.

1) Intermediate code Generation :-

- Intermediate language is used by many compilers ~~from~~ for analyzing and optimizing source program.
- Intermediate language must have two properties.
i) Should be simple and easy to produce.
ii) Should be easy to translate to target program.
- A compiler may produce an explicit intermediate code representing the source program.
- Are generally machine independent.
- Level of intermediate code is close to the level of machine code.

Example :- newval := oldval * ~~12~~ fact + 1

```
id1 := id2 * id3 + 1
temp1 = int2float(12)
temp2 = id2 * id3
temp3 = temp1 + temp2
id1 = temp3
```

2) Code optimization :-

- Involves static analysis of the intermediate code to remove the extraneous operation Thus reducing the size of the program.

- Example:-

```
temp1 = int2float(1)          temp1 = id2 * 1.0
temp2 = id2 * id3      ⟶      id1 = temp1 + id3
temp3 = temp1 + temp2
id1 = temp3
```

3) Code Generation:-

- Involves the translation of optimized code intermediate code into the target language.
- Target language is a relocatable object file containing the machine or assembly codes.

Example:-
```
MOV  id2, R1
MULT id3, R1
ADD  #1, R1
MOV  R1, id1.
```

* Difference between Syntax Analyzer and Lexical Analyzer:-

	Syntax Analyzer	Lexical Analyzer.
(i)	Deals with recursive constructs of a language.	(i) Deals with simple non-recursive construct of a language.
(ii)	Works on smallest meaningful units in a source program to recognize meaningful structure in a programming language.	(ii) Recognizes the smallest meaningful units in a source program and simplifies the job of the syntax analyzer.

TU-2068

Qs* Define tokens, pattern and lexeme with suitable example.

Ans:- Tokens:-

- A pair consisting of a token name and an optional attribute value.
- Token name is an abstract symbol representing a kind of lexical unit e.g. a particular keyword or a sequence of input characters denoting an identifier.

- Pattern:-
- A description of the form that the lexemes of a token may take.
- For a keyword as a token, the pattern is just the sequence of characters that form the keyword.

- Lexeme:-
- A sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instant of that token.

* Annoted Parse Tree:-
- A parse tree constructing for a given input string in which each node showing the value of attribute is called an annoted parse tree.

Eg:

T.U.

V.Imp * Difference between Pascal compiler and C++ compiler.

Ans:-	Pascal Compiler	C++ compiler
(I)	Is one-pass compiler	(I) Is a two-pass compiler.
(II)	Natural languages.	(II) Are symbolic languages.
(III)	It has metaclasses - a unique feature.	(III) It doesn't have metaclasses.
(IV)	Comments are written inside braces.	(IV) Comments are written inside /* ... */

* Code Generator Generators:-
- Based on tree pattern matching and dynamic programming.
-

* Specifications of Tokens:

(1) Alphabets :-
- Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,...,9,A,B,...F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

(2) String :-
- Any finite sequence of alphabets.
- Length of string is the total number of alphabets
- A string with zero/no alphabets / length is known as empty string, denoted by ε (epsilon).

(3) Special Symbols:-
- A typical HLL symbols such as arithmetic symbols, punctuation, assignment, comparison, special assignment, logical specifier, preprocessor, shift operator, logical, etc.

(4) Language :-
- A finite set of strings over an alphabets.
- Finite languages can be described by means of R.E.

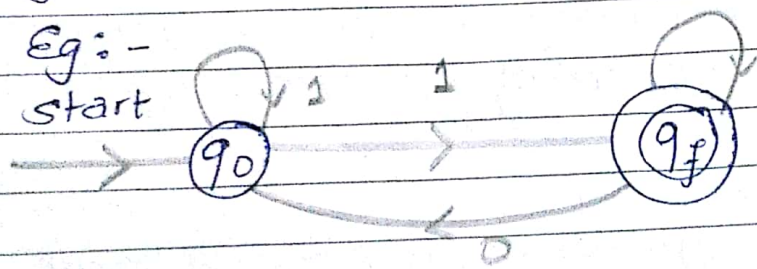
(5) Regular Expressions :-
- Important notation for specifying patterns.
- Expresses the finite languages by defining a pattern for finite strings of symbols.

* Finite Automata :-
- A state machine that takes a string of symbols ~~and~~ as input and changes its

state accordingly.
- A recognizer for regular expression.
- Mathematical model of FA consists of:

i) Finite set of states (Q)
ii) Finite set of input symbols (Σ).
iii) Start state (q_0)
iv) Set of final states (q_f)
v) Transition function (δ) that maps the finite set of states (Q) to a finite set of input symbols (Σ), $Q \times \Sigma \rightarrow Q$.

Eg:-



FA accepting any three binary digits ending with 1